JFP, 35 pages, 2020.
doi:10.1017/xxxxx

On the Correctness of Monadic Backward Induction

NURIA BREDE
Potsdam Institute for Climate Impact Research,
Potsdam, Germany
(e-mail: nuria.brede@pik-potsdam.de)

NICOLA BOTTA
Potsdam Institute for Climate Impact Research,
Potsdam, Germany
(e-mail: botta@pik-potsdam.de)

Abstract

In control theory, to solve a finite-horizon sequential decision problem (SDP) commonly means to find a list of decision rules that result in an optimal expected total reward (or cost) when taking a given number of decision steps. SDPs are routinely solved using Bellman's backward induction (Bellman, 1957). Textbooks typically give more or less formal proofs to show that the backward induction algorithm is correct as solution method for deterministic and stochastic SDPs (Bertsekas, 1995; Puterman, 2014).
In (Botta *et al.*, 2017*a*), Botta, Jansson and Ionescu propose a generic framework for finite horizon, *monadic* SDPs together with a verified monadic version of backward induction for solving such SDPs. In monadic SDPs, the monad captures a generic notion of uncertainty, while a generic measure function aggregates rewards. In the present paper we extend Botta *et al.*'s verification result. Under certain conditions on the measure function, we obtain a correctness result for monadic backward induction that is comparable to textbook correctness proofs for ordinary backward induction. The conditions that we impose are fairly general and can be cast in category-theoretical terms using the notion of Eilenberg-Moore-algebra. They hold for familiar measures like the expected value but also imply that certain measures cannot be used for optimization within the Botta *et al.* framework.
Our development is formalized in Idris (Brady, 2017) as an extension of the Botta *et al.* framework and the sources are available as supplementary material.

1 Introduction

Backward induction is a method introduced by Bellman (1957) that is routinely used to solve *finite-horizon sequential decision problems (SDP)*. Such problems lie at the core of many applications in economics, logistics, and computer science (Finus *et al.*, 2003; Helm, 2003; Heitzig, 2012; Gintis, 2007; Botta *et al.*, 2013; De Moor, 1995, 1999). Examples include inventory, scheduling and shortest path problems, but also the search for optimal strategies in games (Bertsekas, 1995; Diederich, 2001).

In (Botta *et al.*, 2017*a*), Botta, Jansson and Ionescu propose a generic framework for *monadic* finite-horizon SDPs as generalization of the deterministic, non-deterministic and stochastic SDPs treated in control theory textbooks (Bertsekas, 1995; Puterman, 2014). This framework allows to specify such problems and to solve them with a generic version of backward induction that we will refer to as *monadic backward induction*.

The Botta-Jansson-Ionescu-framework, subsequently referred to as *BJI-framework*, *BJI-theory* or simply *framework*, already includes a verification of monadic backward induction with respect to a certain underlying *value* function (see section 3.2). However, in the literature on stochastic SDPs this formulation of the function is itself part of the backward induction algorithm and needs to be verified against an optimization criterion, the *expected total reward*. This raises the question whether monadic backward induction can be considered correct as solution method for the substantially more general monadic SDPs.

In the present paper, we address this question and extend the Botta *et al.* verification result. To this end, we put forward a formal specification that the BJI-value function has to meet. This specification uses an optimization criterion for monadic SDPs that is a generic version of the expected total reward of standard control theory textbooks.[1] We prove that the value function of the BJI-framework meets the specification if the monadic SDP fullfils certain natural conditions. We discuss these conditions and express them in category-theoretical terms using the notion of Eilenberg-Moore-algebra. As corollary we obtain a correctness result for monadic backward induction that can be seen as a generic version of correctness results for standard backward induction like (Bertsekas, 1995, prop. 1.3.1) and (Puterman, 2014, Th. 4.5.1.c).

For the reader unfamiliar with SDPs, we provide a brief informal overview and two simple examples in the next section. We recap the BJI-framework and its (partial) verification result for monadic backward induction in section 3. In section 4 we specify correctness for monadic backward induction and the BJI-value function. We also show that in the general monadic case the value function does not necessarily meet the specification. To resolve this problem, we identify conditions under which the value function does meet the specification. These conditions are stated and discussed in section 5. In section 6 we prove that, given the conditions hold, the BJI-value function and monadic backward induction are correct in the sense defined in section 4. We conclude in section 7.

Throughout the paper we use Idris as our host language (Brady, 2013, 2017). We assume some familiarity with Haskell-like syntax and notions like *functor* and *monad* as used in functional programming. We tacitly consider types as logical statements and programs as proofs, justified by the propositions-as-types correspondence (for an accessible introduction see Wadler, 2015). Our development is formalized in Idris as an extension of a lightweight version of the BJI-framework. The proofs are machine-checked and the source code is available as supplementary material attached to this paper. The sources of this document have been written in literal Idris and are available at (Brede & Botta, 2021), together with some example code. All source files can be type checked with Idris version 1.3.2.

[1] Note that in control theory backward induction is often referred to as *the dynamic programming algorithm* where the term *dynamic programming* is used in the original sense of (Bellman, 1957).

2 Finite-horizon Sequential Decision Problems

In deterministic, non-deterministic and stochastic finite-horizon SDPs, a decision maker seeks to control the evolution of a *(dynamical) system* at a finite number of *decision steps* by selecting certain *controls* in sequence, one after the other. The controls available to the decision maker at a given decision step typically depend on the *state* of the system at that step.

In *deterministic* problems, selecting a control in a state at decision step $t : \mathbb{N}$, determines a unique next state at decision step $t+1$ through a given *transition function*. In *non-deterministic* problems, the transition function yields a whole set of *possible* states at the next decision step. In *stochastic* problems, the transition function yields a *probability distribution* on states at the next decision step.

The notion of *monadic* problem generalizes that of deterministic, non-deterministic and stochastic problem through a transition function that yields an M-structure of next states where M is a monad. For example, the identity monad can be applied to model deterministic systems. Non-deterministic systems can be represented in terms of transition functions that return lists (or some other representations of sets) of next states. Stochastic systems can be represented in terms of probability distribution monads (Giry, 1981; Erwig & Kollmansberger, 2006; Audebaud & Paulin-Mohring, 2009; Jacobs, 2011). The uncertainty monad, the states, the controls and the next function define what is often called a *decision process*.

The idea of sequential decision problems is that each single decision yields a *reward* and these rewards add up to a *total reward* over all decision steps. Rewards are often represented by values of a numeric type, and added up using the canonical addition. If the transition function and thus the evolution of the system is not deterministic, then the resulting possible total rewards need to be aggregated to yield a single outcome value. In stochastic SDPs, evolving the underlying stochastic system leads to a probability distribution on total rewards which is usually aggregated using the familiar expected value measure. The value thus obtained is called the *expected total reward* (Puterman, 2014, ch. 4.1.2). In section 4 we generalize this notion to monadic SDPs as *measured total reward*.

Solving an SDP then consists in *finding a list of rules ps for selecting controls that maximizes the expected (or measured) total reward for n decision steps when starting at decision step t*. This means that when starting from any initial state at decision step t, following the list of rules *ps* for selecting controls will result in a value that is maximal as measure of the sum of rewards along all possible trajectories rooted in that initial state. Equivalently, rewards can instead be considered as *costs* that need to be *minimized*. This dual perspective is taken e.g. in (Bertsekas, 1995). In the subsequent sections we will follow the terminology of the BJI-framework and (Puterman, 2014) and speak of "rewards", but our second example below will illustrate the "cost" perspective.

In mathematical theories of optimal control, the rules for selecting controls are called *policies*. A *policy* for a decision step is simply a function that associates to each possible state a control. As mentioned above, the controls available in a given state typically depend on that state, thus policies are dependently typed functions.

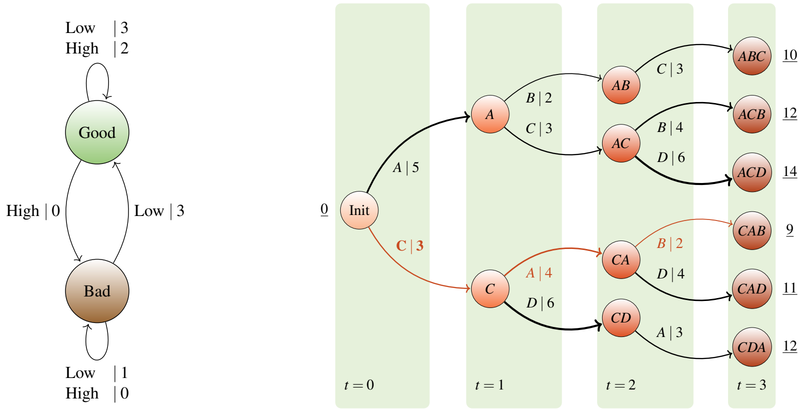
The central idea underlying backward induction is to compute a globally optimal solution of a multi-step SDP incrementally by solving local optimization problems at each decision step. This is captured by *Bellman's principle*: *Extending an optimal policy sequence with an optimal policy yields again an optimal policy sequence*. However, as we will see in section 4.2, one has to carefully check whether for a given SDP backward induction is indeed applicable.

Two features are crucial for finite-horizon, monadic SDPs to be solvable with the BJI-framework that we study in this paper: (1) the number of decision steps has to be given explicitly as input to the backward induction and (2) at each decision step, the number of possible next states has to be *finite*. While (2) is a necessary condition for backward induction to be computable, (1) is a genuine limitation of the framework: in many SDPs, for example in a game of tic-tac-toe, the number of decision steps is bounded from above but not known a priori.

Before we discuss the BJI-framework in the next section, we illustrate the notion of sequential decision problem with two simple examples, one in which the purpose is to maximize rewards and one in which the purpose is to minimize costs. Rewards and costs in these examples are just natural numbers and are summed up with ordinary addition. The first example is a non-deterministic SDP. Although it is somewhat oversimplified, it has the advantage of being tractable for computations by hand while still being sufficient as basis for illustrations in sections 3–5. The second example is a deterministic SDP that stands for the important class of scheduling SDPs. This problem highlights why it is adequate to model state and control spaces by dependent types. As in these simple examples state and control spaces are finite, the transition functions can be described by directed graphs. These are given in fig. 1.

2.1 Example: A toy climate problem

Our first example is a variant of a stochastic climate science SDP studied in (Botta *et al.*, 2018), stripped down to a simple non-deterministic SDP. At every decision step, the world may be in one of two *states*, namely *Good* or *Bad*, and the *controls* determine whether a *Low* or a *High* amount of green house gases is emitted into the atmosphere. If the world is in the *Good* state, choosing *Low* emissions will definitely keep the world in the *Good* state, but the result of choosing high emissions is non-deterministic: either the world may stay in the *Good* or tip to the *Bad* state. Similarly, in the *Bad* state, *High* emissions will definitely keep the world in *Bad*, while with *Low* emissions it might either stay in *Bad* or recover and return to the *Good* state. The transitions just described define a non-deterministic *transition function*. The *rewards* associated with each transition are determined by the respective control and reached state. Now we can formulate an SDP: *"Which policy sequence will maximize the worst-case sum of rewards along all possible trajectories when taking n decisions starting at decision step t?"*. In this simple example the question is not hard to answer: always choose *Low* emissions, independent of decision step and state. The *optimal policy sequence* for any n and t would thus consist of n constant *Low* functions. But in a more realistic example the situation will be more involved: every option will have its benefits and drawbacks encoded in a more complicated reward function, uncertainties



(a) Example 1 (b) Example 2

Fig. 1: Transition graphs for the example SDPs described in section 2. The edge lables denote pairs *control* | *reward* for the associated transitions. In the first example, state and control spaces are constant over time therefore we have omitted the temporal dimension.

might come with different probabilities, there might be intermediate states, different combinations of control options etc. For more along these lines we refer the interested reader to (Botta *et al.*, 2018).

2.2 Example: Scheduling

Scheduling problems serve as canonical examples in control theory textbooks. The one we present here is a slightly modified version of (Bertsekas, 1995, Example 1.1.2).
Think of some machine in a factory that can perform different operations, say A, B, C and D. Each of these operations is supposed to be performed once. The machine can only perform one operation at a time, thus an order has to be fixed in which to perform the operations. Setting the machine up for each operation incurs a specific cost that might vary according to which operation has been performed before. Moreover, operation B can only be performed after operation A has already been completed, and operation D only after operation C. It suffices to fix the order in which the first three operations are to be performed as this uniquely determines which will be the fourth task. The aim is now to choose an order that minimizes the total cost of performing the four operations.

This situation can be modeled as follows as a problem with three decision steps: The *states at each step* are the sequences of operations already performed, with a dummy initial state at step 0. The *controls at a decision step and in a state* are the operations which have not already been performed at previous steps and which are permitted in that state. For example, at decision step 0, only controls A and C are available because of the above constraint on performing B and D. The *transition* and *cost* functions of the problem are depicted by the graph in fig. 1b. As the problem is deterministic, picking a control will result in a unique next state and each sequence of policies will result in a unique trajectory.

In this setting, solving the SDP reduces to finding a control sequence that *minimizes the sum of costs along the single resulting trajectory*. In fig. 1b this is the sequence $CAB(D)$.

3 The BJI-framework

The BJI-framework is a dependently typed formalization of optimal control theory for finite-horizon, discrete-time SDPs. It extends mathematical formulations for stochastic SDPs (Bertsekas, 1995; Bertsekas & Shreve, 1996; Puterman, 2014) to the general problem of optimal decision making under *monadic* uncertainty.
For monadic SDPs, the framework provides a generic implementation of backward induction. It has been applied to study the impact of uncertainties on optimal emission policies (Botta *et al.*, 2018) and is currently used to investigate solar radiation management problems under tipping point uncertainty (TiPES, 2019–2023).

In a nutshell, the framework consists of two sets of components: one for the *specification* of an SDP and one for its *solution* with monadic backward induction.

3.1 Problem specification components

The components for the specification of an SDP consist of global forward declarations. The first one

$M : Type \to Type$

specifies the monad M. As discussed in the previous section, M accounts for the uncertainties that affect the decision problem. For our first example, we could for instance define M to be *List*. For the second example it suffices to use $M = Id$ as the problem is deterministic.

Further, the BJI-framework supports the specification of the *states*, the *controls* and the *transition function* of an SDP through

$$
\begin{aligned}
&X \quad\;\;\, : (t : \mathbb{N}) \to Type \\
&Y \quad\;\;\, : (t : \mathbb{N}) \to X\ t \to Type \\
&next : (t : \mathbb{N}) \to (x : X\ t) \to Y\ t\ x \to M\ (X\ (S\ t))
\end{aligned}
$$

The interpretation is that $X\ t$ represents the states at decision step t.[2] In the first example of section 2, there are just two states (*Good* and *Bad*) such that X is a constant family:

$$
\begin{aligned}
&\mathbf{data}\ State = Good \mid Bad \\
&X\ \_t \qquad\quad = State
\end{aligned}
$$

But in the second example the possible states depend on the decision step t. Taking for example step $t = 2$, we could define

$$
\begin{aligned}
&\mathbf{data}\ State2 = AB \mid AC \mid CA \mid CD \\
&X\ 2 \qquad\qquad = State2
\end{aligned}
$$

Similarly, $Y\ t\ x$ represents the controls available at decision step t and in state x and $next\ t\ x\ y$ represents the states that can be obtained by selecting control y in state x at decision step t. In our first example, the available controls remain constant over time (*High* or

[2] Note that in Idris, S and Z are the familiar constructors of the data type $\mathbb{N}$.

Low) like the states, but for the second example, the type dependency is relevant: e.g. we might define (again at step $t = 2$)

$$\mathbf{data}\ CtrlAC = B \mid D$$
$$Y\ 2\ AC \quad = CtrlAC$$

Recall from section 2 that the monad, the states, the controls and the next function together define a decision process. In order to fully specify a decision problem, one also has to define the rewards obtained at each decision step and the operation that is used to add up rewards. In the BJI-framework, this is done in terms of

$$Val \quad : Type$$
$$reward : (t : \mathbb{N}) \to (x : X\ t) \to Y\ t\ x \to X\ (S\ t) \to Val$$
$$(\oplus) \quad : Val \to Val \to Val$$

Here, *Val* is the type of rewards and $reward\ t\ x\ y\ x'$ is the reward obtained by selecting control y in state x when the next state is x', a value in $X\ (S\ t)$. A few remarks are at place here.

- In many applications, *Val* is a numerical type and controls represent resources (fuel, water, etc.) that come at a cost. In these cases, the reward function encodes the costs and perhaps also the benefits associated with a decision step. Often, the latter also depend both on the current state x and on the next state x'. The BJI-framework nicely copes with all these situations.
- The operation $\oplus$ determines how rewards are added up. It could be a simple arithmetic operation, but it could also be defined in terms of problem-specific parameters, e.g. discount factors to give more weight to current rewards as compared to future rewards.
- Mapping $reward\ t\ x\ y$ onto $next\ t\ x\ y$ (remember that M is a monad and thus a functor) yields a value of type $M\ Val$. These are the *possible* rewards obtained by selecting control y in state x at decision step t. In mathematical theories of optimal control, the implicit assumption often is that *Val* is equal to $\mathbb{R}$ and that the M-structure is a probability distribution on real numbers which can be evaluated with the *expected value* measure. However, in many practical applications, measuring uncertainty of rewards in terms of the expected value is inadequate. The BJI-framework therefore allows the specifications of SDPs in terms of a problem-specific measure.

As just discussed, in SDPs with uncertainty a measure is required to aggregate multiple possible rewards. The BJI-framework supports the specification of the measure through the forward declaration:

$$meas : M\ Val \to Val$$

In our first example we could use the minimum of a list as worst-case measure, while in the second example the measure would just be the identity (as the problem is deterministic).

Before we get to the solution components of the BJI-framework, one more ingredient needs to be specified. In the next section we will formalize a notion of optimality for which it is necessary to be able to compare elements of *Val*. The framework allows users to compare *Val*-values in terms of a problem specific comparison operator:

$$(\sqsubseteq) : Val \to Val \to Type$$

Formalization of example 1:

We use the monoid and preorder structure on $\mathbb{N}$, i.e. $Val = \mathbb{N}$, $(\oplus) = (+)$, $zero = 0$, $(\leqslant) = (\sqsubseteq)$.

$M = List$

Measure:

$$\begin{aligned} &minList : List\ \mathbb{N} \to \mathbb{N} \\ &minList\ [\,] \qquad = 0 \\ &minList\ (x :: [\,]) = x \\ &minList\ (x :: xs) = x\ \text{`}minimum\text{`}\ xs \end{aligned}$$

$meas = minList$

States and Controls:

$$\begin{aligned} &\textbf{data}\ States \quad = Good \mid Bad \\ &\textbf{data}\ Controls = High \mid Low \end{aligned}$$

$$\begin{aligned} &X\ \_t \quad = States \\ &Y\ \_t\ \_x = Controls \end{aligned}$$

Transition function:

$$\begin{aligned} &next\ \_t\ Good\ Low\ = [Good] \\ &next\ \_t\ Bad\ \ High = [Bad] \\ &next\ \_t\ \_x \quad \_y \quad = [Good, Bad] \end{aligned}$$

Rewards:

$$\begin{aligned} &reward\ \_t\ \_x\ Low\ \ Good = 3 \\ &reward\ \_t\ \_x\ High\ Good = 2 \\ &reward\ \_t\ \_x\ Low\ \ Bad \ \ = 1 \\ &reward\ \_t\ \_x\ High\ Bad \ \ = 0 \end{aligned}$$

Fig. 2: A formalization of example 1 from section 2

The operator $(\sqsubseteq)$ is required to define a total preorder on *Val*. In our two examples, we simply have:

$$\begin{aligned} &Val \quad = \mathbb{N} \\ &(\oplus) \quad = (+) \\ &(\sqsubseteq) = (\leqslant) \end{aligned}$$

Three more ingredients are necessary to fully specify a monadic SDP, but we defer discussing them to when they come up in the next subsection. For illustration, a formalization of example 1 can be found in fig. 2. A formalization of example 2 is included in the supplementary material.

3.2 Problem solution components

The second set of components of the BJI-framework is an extension of the mathematical theory of optimal control for stochastic sequential decision problems to monadic problems. Here, we provide a summary of the theory. Motivation and full details can be found in (Botta *et al.*, 2017*b*,a, 2018).

The theory formalizes the notions of policy (decision rule) from section 2 as

$$\begin{aligned} &Policy \quad : (t : \mathbb{N}) \to Type \\ &Policy\ t = (x : X\ t) \to Y\ t\ x \end{aligned}$$

Policy sequences of length $n : \mathbb{N}$ are then essentially vectors of policies[3]

$$\begin{aligned} &\textbf{data}\ PolicySeq : (t, n : \mathbb{N}) \to Type\ \textbf{where} \\ &\quad Nil : \{t : \mathbb{N}\} \to PolicySeq\ t\ Z \\ &\quad (::) : \{t, n : \mathbb{N}\} \to Policy\ t \to PolicySeq\ (S\ t)\ n \to PolicySeq\ t\ (S\ n) \end{aligned}$$

Notice the role of the step (time) index t and of the length index n in the constructors of policy sequences: as for plain vectors and lists, prepending a policy to a policy sequence of length n yields a policy sequence of length $n+1$. As one would expect, policies for taking

[3] The curly brackets in the types of *Nil* and $(::)$ indicate that t and n are implicit arguments.

decisions at step t can only be prepended to policy sequences for taking *first* decisions at step $t+1$ and the operation yields policy sequences for taking *first* decisions at step t!

The perhaps most important ingredient of backward induction is a *value function* that incrementally measures and adds up rewards. For a given decision problem, the value function takes two arguments: a policy sequence ps for making n decision steps starting from decision step t and an initial state $x : X\ t$. It computes the value of taking n decision steps according to the policies ps when starting in x. In the BJI-framework, the value function is defined as

$$
\begin{array}{l}
val : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to X\ t \to Val \\
val\ \{t\}\ Nil\ x \qquad\quad = zero \\
val\ \{t\}\ (p :: ps)\ x = \textbf{let}\ y \quad\ = p\ x\ \textbf{in} \\
\qquad\qquad\qquad\qquad\ \textbf{let}\ mx' = next\ t\ x\ y\ \textbf{in} \\
\qquad\qquad\qquad\qquad\ meas\ (map\ (reward\ t\ x\ y \bigoplus val\ ps)\ mx')
\end{array}
$$

Notice that, independently of the initial state x, the value of the empty policy sequence is *zero*. This is a problem-specific reference value

$$zero : Val$$

that has to be provided as part of a problem specification.[4] It is one of the specification components that we have not discussed in section 3.1. The value of a policy sequence consisting of a first policy p and of a tail policy sequence ps is defined inductively as the measure of an M-structure of Val-values. These values are obtained by first computing the control y dictated by p in x, the M-structure of possible next states mx' dictated by *next* and finally by adding $reward\ t\ x\ y\ x'$ and $val\ ps\ x'$ for all x' in mx'. The result of this functorial mapping is then measured with the problem-specific measure *meas* to obtain a result of type Val. The function which is mapped onto mx' is just a lifted version of $\oplus$:

$$
\begin{array}{l}
(\bigoplus) : \{A : Type\} \to (f, g : A \to Val) \to A \to Val \\
f \bigoplus g = \lambda a \Rightarrow f\ a \oplus g\ a
\end{array}
$$

We will come back to the value function of the BJI-theory in section 4 where we will contrast it with a function val' that, for a policy sequence ps and an initial state x, computes the measure of the sum of the rewards along all possible trajectories starting at x under ps (the *measured total reward* that we anticipated in section 2). For the time being, though, we accept the notion of value of a policy sequence as put forward in the BJI-theory and we show how the definition of *val* can be employed to compute policy sequences that are provably optimal in the sense of

$$
\begin{array}{l}
OptPolicySeq : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to Type \\
OptPolicySeq\ \{t\}\ \{n\}\ ps = (ps' : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps'\ x \sqsubseteq val\ ps\ x
\end{array}
$$

Notice the universal quantification in this definition: A policy sequence ps is defined to be optimal iff $val\ ps'\ x \sqsubseteq val\ ps\ x$ for any policy sequence ps' and for any state x.

The generic implementation of backward induction in the BJI-framework is an application of *Bellman's principle of optimality* mentioned in section 2. In control theory

[4] The name might suggest that *zero* is supposed to be a neutral element relative to $\oplus$. However, this is not required by the framework.

textbooks, this principle is often referred to as *Bellman's equation*. It can be suitably formulated in terms of the notion of *optimal extension*. We say that a policy $p : Policy\ t$ is an optimal extension of a policy sequence $ps : Policy\ (S\ t)\ n$ if it is the case that the value of $p :: ps$ is at least as good as the value of $p' :: ps$ for any policy p' and for any state $x : X\ t$:

$$\begin{aligned}
&OptExt : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t \to Type\\
&OptExt\ \{t\}\ ps\ p = (p' : Policy\ t) \to (x : X\ t) \to val\ (p' :: ps)\ x \sqsubseteq val\ (p :: ps)\ x
\end{aligned}$$

With the notion of optimal extension in place, Bellman's principle can be formulated as

$$\begin{aligned}
&Bellman : \{t, n : \mathbb{N}\} \to\\
&\qquad (ps : PolicySeq\ (S\ t)\ n) \to OptPolicySeq\ ps \to\\
&\qquad (p\ \ : Policy\ t) \to OptExt\ ps\ p \to\\
&\qquad OptPolicySeq\ (p :: ps)
\end{aligned}$$

In words: *extending an optimal policy sequence with an optimal extension (of that policy sequence) yields an optimal policy sequence* or shorter *prefixing with optimal extensions preserves optimality*. Proving Bellman's optimality principle is almost straightforward and crucially relies on $\sqsubseteq$ being reflexive and transitive (remember that, in the BJI-framework, $\sqsubseteq$ is required to be a total preorder). The proof obligation is to show that

$$val\ (p' :: ps')\ x \sqsubseteq val\ (p :: ps)\ x$$

for arbitrary p', ps' and x of suitable types. This is achieved by transitivity of $\sqsubseteq$ on two subproofs: $val\ (p' :: ps')\ x \sqsubseteq val\ (p' :: ps)\ x$ and $val\ (p' :: ps)\ x \sqsubseteq val\ (p :: ps)\ x$. The second inequality directly follows from the last argument of *Bellman*, a proof that $OptExt\ ps\ p$. The first inequality follows from the optimality of *ps* (the second argument of *Bellman*), reflexivity of $\sqsubseteq$ and two crucial *monotonicity* properties:

$$\begin{aligned}
&plusMonSpec\ \ : \{v1, v2, v3, v4 : Val\} \to v1 \sqsubseteq v2 \to v3 \sqsubseteq v4 \to (v1 \oplus v3) \sqsubseteq (v2 \oplus v4)\\
&measMonSpec : \{A : Type\} \to (f, g : A \to Val) \to ((a : A) \to f\ a \sqsubseteq g\ a) \to\\
&\qquad\qquad\qquad (ma : M\ A) \to meas\ (map\ f\ ma) \sqsubseteq meas\ (map\ g\ ma)
\end{aligned}$$

The second condition is a special case of the measure monotonicity requirement originally formulated by Ionescu in (Ionescu, 2009) in the context of a theory of vulnerability and monadic dynamical systems. It is a natural property and the expected value measure, the worst (best) case measure and any sound statistical measure fulfill it. Like the reference value *zero* discussed above, *plusMonSpec* and *measMonSpec* are specification components of the BJI-framework that we have not discussed in section 3.1. We provide a proof of *Bellman* in appendix 5. As one would expect, the proof makes essential use of the recursive definition of the function *val* discussed above. As a consequence, this precise definition of *val* plays a crucial role for the verification of backward induction in (Botta *et al.*, 2017*a*).

Apart from the increased level of generality, the definition of *val* and *Bellman* are in fact just an Idris formalization of Bellman's equation as formulated in control theory textbooks. With *Bellman* and provided that we can compute optimal extensions of arbitrary policy sequences

$$\begin{aligned}
&optExt\qquad : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t\\
&optExtSpec : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to OptExt\ ps\ (optExt\ ps)
\end{aligned}$$

it is easy to derive an implementation of monadic backward induction that computes provably optimal policy sequences with respect to *val*: first, notice that the empty policy sequence is optimal:

$$\begin{aligned}&\mathit{nilOptPolicySeq} : \mathit{OptPolicySeq}\ \mathit{Nil}\\&\mathit{nilOptPolicySeq}\ \mathit{Nil}\ x = \mathit{reflexive}\ \mathit{lteTP}\ \mathit{zero}\end{aligned}$$

This is the base case for constructing optimal policy sequences by backward induction, starting from the empty policy sequence:

$$\begin{aligned}&\mathit{bi} : (t, n : \mathbb{N}) \to \mathit{PolicySeq}\ t\ n\\&\mathit{bi}\ t\ Z \quad\quad = \mathit{Nil}\\&\mathit{bi}\ t\ (S\ n) = \mathbf{let}\ \mathit{ps} = \mathit{bi}\ (S\ t)\ n\ \mathbf{in}\ \mathit{optExt}\ \mathit{ps} :: \mathit{ps}\end{aligned}$$

That *bi* computes optimal policy sequences with respect to *val* is then proved by induction on n:

$$\begin{aligned}&\mathit{biOptVal} : (t, n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)\\&\mathit{biOptVal}\ t\ Z \quad\quad = \mathit{nilOptPolicySeq}\\&\mathit{biOptVal}\ t\ (S\ n) = \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}\ \mathbf{where}\\&\quad \mathit{ps} \ \ : \mathit{PolicySeq}\ (S\ t)\ n \quad ; \quad \mathit{ps} \ \ = \mathit{bi}\ (S\ t)\ n\\&\quad \mathit{ops} : \mathit{OptPolicySeq}\ \mathit{ps} \quad ; \quad \mathit{ops} = \mathit{biOptVal}\ (S\ t)\ n\\&\quad p \ \ \ : \mathit{Policy}\ t \quad\quad\quad\quad ; \quad p \ \ \ = \mathit{optExt}\ \mathit{ps}\\&\quad \mathit{oep} : \mathit{OptExt}\ \mathit{ps}\ p \quad\quad ; \quad \mathit{oep} = \mathit{optExtSpec}\ \mathit{ps}\end{aligned}$$

This is the verification result for *bi* of (Botta *et al.*, 2017*a*).[5]

3.3 BJI-framework wrap-up.

The specification and solution components discussed in the last two sections are all we need to formulate precisely the problem of correctness for the BJI-framework. This is done in the next section. Before turning to it, two final remarks are necessary:

- The theory proposed in (Botta *et al.*, 2017*a*) is slightly more general than the one summarized above. Here, policies are just functions from states to controls:

  $$\begin{aligned}&\mathit{Policy} : (t : \mathbb{N}) \to \mathit{Type}\\&\mathit{Policy}\ t = (x : X\ t) \to Y\ t\ x\end{aligned}$$

  By contrast, in (Botta *et al.*, 2017*a*), policies are indexed over a number of decision steps n

  $$\begin{aligned}&\mathit{Policy} : (t, n : \mathbb{N}) \to \mathit{Type}\\&\mathit{Policy}\ t\ Z \quad\quad = \mathit{Unit}\\&\mathit{Policy}\ t\ (S\ m) = (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ (S\ m)\ x \to \mathit{GoodCtrl}\ t\ x\ m\end{aligned}$$

  and their domain for $n > 0$ is restricted to states that are *reachable* and *viable* for n steps. This allows the (Botta *et al.*, 2017*a*) version to cope with states whose control set is empty and with transition functions that return empty M-structures of next states.

[5] Note that *biOptVal* is called *biLemma* in (Botta *et al.*, 2017*a*). We chose the new name to emphasize that *bi* computes optimal policy sequences with respect to *val*.

This generality, however, comes at a cost: Compare e.g. the proof of Bellman's principle from the last subsection with the corresponding proof in (Botta *et al.*, 2017*a*, Appendix B). The impact of the reachability and viability constraints on other parts of the theory is even more severe.
Here, we have decided to trade some generality for better readability and opted for a simplified version of the original theory. The price to pay for this simplification is that we have to explicitly require controls to be non-empty:

```
notEmptyY : (t : ℕ) → (x : X t) → Y t x
```

We also impose a non-emptiness requirement on the transition function *next* that will be discussed in section 5.3.

```
nextNotEmpty : {t : ℕ} → (x : X t) → (y : Y t x) → NotEmpty (next t x y)
```

- In section 3.2, we have not discussed under which conditions one can implement optimal extensions of arbitrary policy sequences. This is an interesting topic that is however orthogonal to the purpose of the current paper. For the same reason we have not addressed the question of how to make *bi* more efficient by tabulation. We briefly discuss the specification and implementation of optimal extensions in the BJI-framework in appendix 7. We refer the reader interested in tabulation of *bi* to SequentialDecisionProblems.TabBackwardsInduction of (Botta, 2016–2021).

4 Correctness for monadic backward induction

In this section we formally specify the notions of correctness for monadic backward induction *bi* and the value function *val* of the BJI-framework that we will study in the remainder of this paper. We develop these notions as generic variants of the corresponding notions for stochastic SDPs.

4.1 Extension of the BJI-framework

In the previous section, we have seen that a monadic SDP can be specified in terms of nine components: M, X, Y, *next*, *Val*, *zero*, $\oplus$, $\sqsubseteq$ and *reward*.

Given a policy sequence (optimal or not) and an initial state for an SDP, we can compute the M-structure of possible trajectories starting at that state:

```
data StateCtrlSeq : (t, n : ℕ) → Type where
  Last : {t : ℕ} → X t → StateCtrlSeq t (S Z)
  (∗∗) : {t, n : ℕ} → Σ (X t) (Y t) → StateCtrlSeq (S t) (S n) → StateCtrlSeq t (S (S n))

trj : {t, n : ℕ} → PolicySeq t n → X t → M (StateCtrlSeq t (S n))
trj {t} Nil x      = pure (Last x)
trj {t} (p :: ps) x = let y = p x in
                      let mx' = next t x y in
                      map ((MkSigma x y)∗∗) (mx' >>= trj ps)
```

where we use *StateCtrlSeq* as type of trajectories. Furthermore, we can compute the *total reward* for a single trajectory, i.e. its sum of rewards:

$$
\begin{aligned}
&\mathit{sumR} : \{t, n : \mathbb{N}\} \to \mathit{StateCtrlSeq}\ t\ n \to \mathit{Val}\\
&\mathit{sumR}\ \{t\}\ (\mathit{Last}\ x) \qquad\qquad\quad = \mathit{zero}\\
&\mathit{sumR}\ \{t\}\ (\mathit{MkSigma}\ x\ y \mathbin{{*}{*}} \mathit{xys}) = \mathit{reward}\ t\ x\ y\ (\mathit{head}\ \mathit{xys}) \oplus \mathit{sumR}\ \mathit{xys}
\end{aligned}
$$

where *head* ist the helper function

$$
\begin{aligned}
&\mathit{head} : \{t, n : \mathbb{N}\} \to \mathit{StateCtrlSeq}\ t\ (S\ n) \to X\ t\\
&\mathit{head}\ (\mathit{Last}\ x) \qquad\qquad\quad = x\\
&\mathit{head}\ (\mathit{MkSigma}\ x\ y \mathbin{{*}{*}} \mathit{xys}) = x
\end{aligned}
$$

By mapping *sumR* onto an M-structure of trajectories, we obtain an M-structure containing the individual sums of rewards of the trajectories. Now, using the measure function, we can compute the generic analogue of the expected total reward for a policy sequence *ps* and an initial state x:

$$
\begin{aligned}
&\mathit{val}' : \ \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ t\ n) \to (x : X\ t) \to \mathit{Val}\\
&\mathit{val}'\ \mathit{ps} = \mathit{meas} \circ \mathit{map}\ \mathit{sumR} \circ \mathit{trj}\ \mathit{ps}
\end{aligned}
$$

As anticipated in section 2 we call the value computed by val' the *measured total reward*. Recall that solving a stochastic SDP commonly means finding a policy sequence that maximizes the *expected total reward*. By analogy, define that solving a monadic SDP means to to find a policy sequence that maximizes the *measured total reward*. I.e. given t and n, the solution of a monadic SDP is a sequence of n policies that maximizes the measure of the sum of rewards along all possible trajectories of length n that are rooted in an initial state at step t.

Again by analogy to the stochastic case, we define monadic backward induction to be correct, if for a given SDP the policy sequence computed by *bi* is the solution to the SDP. I.e., we consider *bi* to be correct if it meets the specification

$$
\mathit{biOptMeasTotalReward} : (t, n : \mathbb{N}) \to \mathit{GenOptPolicySeq}\ \mathit{val}'\ (\mathit{bi}\ t\ n)
$$

where *GenOpPolicySeq* is a generalized version of the optimality predicate *OptPolicySeq* from section 3.2

$$
\begin{aligned}
&\mathit{GenOptPolicySeq} : \{t, n : \mathbb{N}\} \to (\mathit{PolicySeq}\ t\ n \to X\ t \to \mathit{Val}) \to \mathit{PolicySeq}\ t\ n \to \mathit{Type}\\
&\mathit{GenOptPolicySeq}\ \{t\}\ \{n\}\ f\ \mathit{ps} = (\mathit{ps}' : \mathit{PolicySeq}\ t\ n) \to (x : X\ t) \to f\ \mathit{ps}'\ x \sqsubseteq f\ \mathit{ps}\ x
\end{aligned}
$$

As recapitulated in section 3.2, Botta *et al.* have already shown that if M is a monad, $\sqsubseteq$ a total preorder and $\oplus$ and *meas* fulfill two monotonicity conditions, then $\mathit{bi}\ t\ n$ yields an optimal policy sequence with respect to the value function *val* in the sense that $\mathit{val}\ \mathit{ps}'\ x \sqsubseteq \mathit{val}\ (\mathit{bi}\ t\ n)\ x$ for any policy sequence ps' and initial state x, for arbitrary $t, n : \mathbb{N}$. Or, expressed using the generalized optimality predicate,

$$
\mathit{GenOptPolicySeq}\ \{t\}\ \{n\}\ \mathit{val}\ (\mathit{bi}\ t\ n)
$$

As seen in section 3.2, the function *val* measures and adds rewards incrementally. But does it always compute the measured total reward like val'? Modulo differences in the presentation (Puterman, 2014, Theorem 4.2.1) suggests that for standard stochastic SDPs, *val* and val' are extensionally equal, which in turn allows the use of backward induction for solving these SDPs. Generalizing, we therefore consider *val* as correct if it fulfills the specification

$$
\mathit{valMeasTotalReward} : \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ t\ n) \to (x : X\ t) \to \mathit{val}\ \mathit{ps}\ x = \mathit{val}'\ \mathit{ps}\ x
$$

If this equality held for the general monadic SDPs of the BJI-theory, we could prove the correctness of *bi* as immediate corollary of *valMeasTotalReward* and Botta *et al.* 's result *biOptVal*. The statement *biOptMeasTotalReward* can be seen as a generic version of textbook correctness statements for backward induction as solution method for stochastic SDPs like (Bertsekas, 1995, prop.1.3.1) or (Puterman, 2014, Theorem 4.5.1.c). By proving *valMeasTotalReward* we could therefore extend the verification of (Botta *et al.*, 2017*a*) and obtain a stronger correctness result for monadic backward induction.

Our main objective in the remainder of the paper is therefore to prove that *valMeasTotalReward* holds. But there is a problem.

4.2 The problem with the BJI-value function

A closer look at *val* and *val′* reveals two quite different computational patterns: applied to a policy sequence *ps* of length $n+1$ and a state *x*, the function *val* directly evaluates *meas* on the *M*-structure of rewards corresponding to the possible next states after one step. This entails further evaluations of *meas* for each possible next state. By contrast, *val′ ps x* entails only one evaluation of *meas*, independently of the length of *ps*. The computation, however, builds up an intermediate *M*-structure of state-control sequences. The elements of this *M*-structure, the state-control sequences, are then consumed by *sumR* and finally the *M*-structure of rewards is reduced by *meas*.

For illustration, let us revisit our first example from section 2 as formalized in fig. 2. To do an example calculation with *val* and *val′* we first need a concrete policy sequence as input. The simplest two policies are the two constant policies:

$$constH : (t : \mathbb{N}) \to Policy\ t$$
$$constH\ \_t = const\ High$$
$$constL : (t : \mathbb{N}) \to Policy\ t$$
$$constL\ \_t = const\ Low$$

From these, we can define a policy sequence

$$ps : PolicySeq\ 0\ 3$$
$$ps = constH\ 0 :: (constL\ 1 :: (constH\ 2 :: Nil))$$

It is instructive to compute *val ps Good* and *val′ ps Good* by hand. Without instantiating the measure *meas* for the moment, the computations roughly exhibit the following structure:

$$val\ ps\ Good = meas\ [2 + meas\ [3 + meas\ [2, 0]], 0 + meas\ [3 + meas\ [2, 0], 1 + meas\ [0]]]$$
$$val'\ ps\ Good = meas\ [7, 5, 5, 3, 1]$$

It is not “obviously clear” that *val* and *val′* are extensionally equal. In the deterministic case, i.e. for $M = Id$ and $meas = id$, they are indeed equal without imposing any further conditions (as we will see in section 6). For the stochastic case, (Puterman, 2014, Theorem 4.2.1) suggests that the equality should hold. But for the monadic case, no such result has been established. And as it turns out, in general the functions *val* and *val′* are not unconditionally equal – consider the following counter-example: We stay in the setting of example 1, but use as measure the plain arithmetic sum

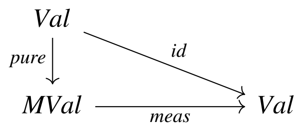
$$meas = foldr\ (+)\ 0$$

This measure fulfills the *measMonSpec* condition (section 3.2) imposed by the BJI-framework. But if we instantiate the above computations with it, then we get $val\ ps\ Good = 13$ and $val'\ ps\ Good = 21$! We thus see that the equality between *val* and *val'* cannot hold unconditionally in the generic setting of the BJI-framework. In the next section we therefore present conditions under which the equality *does* hold.

5 Correctness conditions

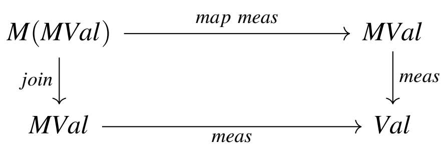
We now formulate three conditions on measure functions that imply the extensional equality of *val* and *val'*:
The measure needs to be left-inverse to *pure*: [6]

(1) $measPureSpec : meas \circ pure \doteq id$



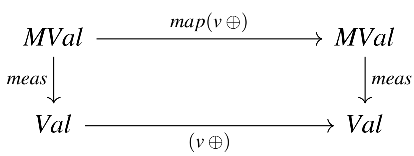
Applying the measure after *join* needs to be extensionally equal to applying it after *map meas*:

(2) $measJoinSpec : meas \circ join \doteq meas \circ map\ meas$



For arbitrary $v : Val$ and non-empty $mv : M\ Val$ applying the measure after mapping $(v\oplus)$ onto mv needs to be equal to applying $(v\oplus)$ after the measure:

(3) $measPlusSpec : (v : Val) \to (mv : M\ Val) \to (NotEmpty\ mv) \to$
$(meas \circ map\ (v\oplus))\ mv = ((v\oplus) \circ meas)\ mv$



[6] The symbol $\doteq$ denotes *extensional* equality, see appendix 1.2

Essentially, these conditions assure that the measure is well-behaved relative to the monad structure and the $\oplus$-operation.

5.1 Examples and counter-examples

To get a better intuition, let us consider a few measures that do or do not fulfill the three conditions. Simple examples of admissible measures are the minimum (*minList* as defined in fig. 2) or maximum ($maxList = foldr$ ‘$maximum$‘ 0) of a list for $M = List$ with $\mathbb{N}$ as type of values and ordinary addition as $\oplus$. It is straightforward to prove that the conditions hold for these two measures and the proofs for *maxList* are included in the supplementary material.

As to counter-examples, let's take another look at our counter-example from the last section, the arithmetic sum of a list. It does fulfill *measPureSpec* and *measJoinSpec*, the first by definition, the second by structural induction using the associativity of $+$ (the list monads' *join* is *concat*). But it fails to fulfill *measPlusSpec*. The premiss *nonEmpty mv* (see appendix 1.2 and next subsection) tells us that the list must not be $[\,]$ – otherwise we can prove the equality from a contradiction. But if the list has the form $a :: as$ we would have to show the equality:

$$(sum \circ map\ (v+))\ (a :: as) = ((v+) \circ sum)\ (a :: as)$$

Clearly, if $v \neq 0$ and $as \neq [\,]$ this equality cannot hold. This is why in the last section the equality of *val* and *val'* failed for $meas = sum$. A similar failure would arise if we chose $meas = foldr\ (*)\ 1$ instead, as $+$ does not distribute over $*$. But if we turned the situation around by setting $\oplus = *$ and $meas = sum$, the condition *measPlusSpec* would hold thanks to the usual arithmetic distributivity law for $*$ over $+$.

What about the other conditions? We remain in the setting of example 1 as above, and just vary the measure. Using a somewhat contrived variation of *maxList*

$$meas = foldr\ (\lambda x, v \Rightarrow (x + 1\ \text{‘}maximum\text{‘}\ v))\ 0$$

it suffices to consider for an arbitrary $n : \mathbb{N}$ that

$$(meas \circ pure)\ n = meas\ [n] = (n+1)\ \text{‘}maximum\text{‘}\ 0 = n + 1 \neq n = id\ n$$

to see that now the condition *measPureSpec* fails. To exhibit a measure that fails the condition *measJoinSpec*, we switch to *Double* as type of values (still with addition as binary operator) and the arithmetic average as measure $meas = avg$. Taking a list of lists of different lengths like [[1], [2, 3]] we have

$$\begin{aligned} meas\ (join\ [[1],[2,3]]) &= avg\ [1,2,3] = 2 \\ &\neq \\ meas\ (map\ meas\ [[1],[2,3]]) &= avg\ [1,2.5] \ = 1.75 \end{aligned}$$

All of the measures considered in this subsection do fulfill the *measMonSpec* condition imposed by the BJI-theory. This raises the question how previously admissible measures are impacted by adding the three new conditions to the framework.

5.2 Impact on previously admissible measures

As we have seen in section 3.2, the BJI-framework already requires measures to fulfill the monotonicity condition

$$\mathit{measMonSpec} : \{A : \mathit{Type}\} \to (f, g : A \to \mathit{Val}) \to ((a : A) \to (f\ a) \sqsubseteq (g\ a)) \to \\ (ma : M\ A) \to \mathit{meas}\ (\mathit{map}\ f\ ma) \sqsubseteq \mathit{meas}\ (\mathit{map}\ g\ ma)$$

Botta *et al.* show that the arithmetic average (for $M = \mathit{List}$), the worst-case measure (for $M = \mathit{List}$ and for $M = \mathit{Prob}$) and the expected value measure (for $M = \mathit{Prob}$) all fulfill *measMonSpec*. Thus, a natural question is whether these measures also fulfill the three additional requirements.

Expected value measure. Most applications of backward induction aggregate possible rewards with the expected value measure. In a nutshell, for a numerical type Q, the expected value of a probability distribution on Q is

$$\mathit{expVal} : \mathit{Num}\ Q \Rightarrow \mathit{Prob}\ Q \to Q \\ \mathit{expVal}\ \mathit{spq} = \mathit{sum}\ [q * \mathit{prob}\ \mathit{spq}\ q \mid q \leftarrow \mathit{supp}\ \mathit{spq}]$$

where *prob* and *supp* are generic functions that encode the notion of *probability* and of *support* associated with a finite probability distribution:

$$\mathit{prob} : \{A : \mathit{Type}\} \to \mathit{Prob}\ A \to A \to Q \\ \mathit{supp} : \{A : \mathit{Type}\} \to \mathit{Prob}\ A \to \mathit{List}\ A$$

For *spa* and *a* of suitable types, *prob spa a* represents the probability of *a* according to *spa*. Similarly, *supp spa* returns a list of those values whose probability is not zero in *spa*. The probability function *prob* has to fulfill the axioms of probability theory. In particular,

$$\mathit{sum}\ [\mathit{prob}\ \mathit{spa}\ a \mid a \leftarrow \mathit{supp}\ \mathit{spa}] = 1$$

This condition implies that probability distributions cannot be empty, a precondition of *measPlusSpec*. Putting forward minimal specifications for *prob* and *supp* is not completely trivial but if the $+$-operation associated with Q is commutative and associative, if $*$ distributes over $+$ and if the *map* and *join* associated with *Prob* – for f, a, b, *spa* and *spspa* of suitable types – fulfill the conservation law

$$\mathit{prob}\ (\mathit{map}\ f\ \mathit{spa})\ b = \mathit{sum}\ [\mathit{prob}\ \mathit{spa}\ a \mid a \leftarrow \mathit{supp}\ \mathit{spa}, f\ a == b]$$

and the total probability law

$$\mathit{prob}\ (\mathit{join}\ \mathit{spspa})\ a = \mathit{sum}\ [\mathit{prob}\ \mathit{spa}\ a * \mathit{prob}\ \mathit{spspa}\ \mathit{spa} \mid \mathit{spa} \leftarrow \mathit{supp}\ \mathit{spspa}]$$

then the expected value measure fulfills *measPureSpec*, *measJoinSpec* and *measPlusSpec*. This is not surprising as it is standard to apply backward induction to SDPs with the expected value of possible rewards as measure.

Average and arithmetic sum. As can already be concluded from the corresponding counter-examples in the previous subsection, neither the plain arithmetic average nor the arithmetic sum are suited as measure when using the standard monad structure on *List* to represent non-deterministic uncertainty. However, the arithmetic average can be used as measure, if *List* is endowed with another monad structure, corresponding to a probability monad for uniform distributions.

Worst-case measures. In many important applications in climate impact research but also in portfolio management and sports, decisions are taken as to minimize the consequences of worst case outcomes. Depending on how "worse" is defined, the corresponding measures might pick the maximum or minimum from an M-structure of values. In the previous subsection we considered an example in which the monad was *List*, the operation $\oplus$ plain addition together with either *maxList* or *minList* as measure. And indeed we can prove that for both measures the three requirements hold (the proofs for *maxList* can be found in the supplementary material). This gives us a notion of worst-case measure that is admissible for monadic backward induction.

We can thus conclude that the new requirements hold for certain familiar measures, but also that they have non-trivial consequences on measures that can be used in the BJI-framework.

5.3 The measure conditions from an abstract perspective

Now that we have seen what the three conditions mean for concrete examples, we can consider them from a more abstract point of view.

Category-theoretical perspective. Readers familiar with the theory of monads might have recognized that the first two conditions ensure that *meas* is the structure map of a monad algebra for M on *Val* and thus the pair $(Val, meas)$ is an object of the Eilenberg-Moore category associated with the monad M. The third condition requires the map $(v\oplus)$ to be an M-algebra homomorphism – a structure preserving map – for arbitrary values v.

This perspective allows us to use existing knowledge about monad algebras as a first criterion for choosing measures. For example, the Eilenberg-Moore-algebras of the list monad are monoids – this implicitly played a role in the examples we considered above. Jacobs (2011) shows that the algebras of the distribution monad for probability distribution with finite support correspond to convex sets. Interestingly, convex sets play an important role in the theory of optimal control (Bertsekas *et al.*, 2003).

Measures for the list monad. The knowledge that monoids are *List*-algebras suggests a generic description of admissible measures for $M = List$: Given a monoid $(Val, \odot, b)$, we can prove that monoid homomorphisms of the form *foldr* $\odot$ b fulfill the three conditions, if $\oplus$ distributes over $\odot$ on the left. I.e. for $meas = foldr\ \odot\ b$ the three conditions can be proven from

$$
\begin{array}{lllll}
odotNeutrRight & : (l : Val) & \to l \odot neutr & = l \\
odotNeutrLeft & : (r : Val) & \to neutr \odot r & = r \\
odotAssociative & : (l, v, r : Val) & \to l \odot (v \odot r) & = (l \odot v) \odot r \\
oplusOdotDistrLeft & : (n, l, r : Val) & \to n \oplus (l \odot r) & = (n \oplus l) \odot (n \oplus r)
\end{array}
$$

Neutrality of b on the right is needed for *measPureSpec*, while *measJoinSpec* follows from neutrality on the left and the associativity of $\odot$. The algebra morphism condition on $(v\oplus)$

is provable from the distributivity of $\oplus$ over $\odot$ and again neutrality of b on the right. If moreover $\odot$ is monotone with respect to $\sqsubseteq$

$$odotMon : \{a, b, c, d : Val\} \to a \sqsubseteq b \to c \sqsubseteq d \to (a \odot c) \sqsubseteq (b \odot d)$$

then we can also prove *measMonSpec* using the transitivity of $\sqsubseteq$. The proofs are simple and can be found in the supplementary material to this paper. This also illustrates how the three abstract conditions follow from more familiar algebraic properties.

Mutual independence. Although it does not seem surprising, it should be noted that the three conditions are mutually independent. This can be concluded from the counter-examples in section 5.1: The sum, the modified list maximum and the arithmetic average each fail exactly one of the three conditions.

Sufficient vs. necessary. The three conditions are sufficient to prove the extensional equality of the functions *val* and *val′*. They are justified by their level of generality and the fact that they hold for standard *measures* used in control theory. However, we leave open the interesting question whether these conditions are also necessary for the correctness of monadic backward induction.

Non-emptiness requirement. Note that *mv* in the premisses of *measPlusSpec* is required to be non-empty. This condition makes sense: If we use again the list monad with $Val = \mathbb{N}$ and $\oplus = +$, it is not hard to see that for any natural number n greater than 0 the equality $meas\ (map\ (n+)\ [\,]) = n + meas\ [\,]$ must fail. Thus, the only way to prove the base case of *measPlusSpec* is by contradiction with the non-emptiness premiss.

However, omitting the premiss $mv : NotEmpty$ would not prevent us from generically proving the correctness result in the next section – it would even simplify matters as it would spare us reasoning about preservation of non-emptiness. But it would implicitly restrict the class of monads that can be used to instantiate M. For example, we have seen above, that *measPlusSpec* is not provable for the empty list without the non-emptiness premiss. We would therefore need to resort to a type of non-empty lists instead.

The price to pay for including the non-emptiness premiss is the additional condition *nextNotEmpty* on the transition function *next* that was already stated in section 3.3. Moreover, we have to postulate non-emptiness preservation laws for the monad operations (appendix 1.2) and to prove an additional lemma about the preservation of non-emptiness (appendix 4). Conceptually, it might seem cleaner to omit the non-emptiness condition: In this case, the remaining conditions would only concern the interaction between the monad, the measure, the type of values and the binary operation $\oplus$. However, the non-emptiness preservation laws seem less restrictive with respect to the monad. In particular, for our above example of ordinary lists they hold (the relevant proofs can be found in the supplementary material). Thus we have opted for explicitly restricting the *next* function instead of implicitly restricting the class of monads for which the results of section 6 holds.

Given the three conditions *measPureSpec*, *measJoinSpec*, *measPlusSpec* on the measure function hold, we can prove the extensional equality of the functions *val* and *val′* generically. This is what we will do in the next section.

6 Correctness Proofs

In this section we show that *val* (section 3.2) and *val′* (section 4) are extensionally equal

$$valMeasTotalReward : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val'\ ps\ x = val\ ps\ x$$

given the measure requirements from the previous section hold. As a corollary we then obtain our correctness result for monadic backward induction.

We can understand the proof of *valMeasTotalReward* as an optimizing program transformation from the less efficient but "obviously correct" implementation *val′* to the more efficient implementation *val*. Therefore the equational reasoning proofs in this section will proceed from *val′* to *val*. In section 5 we have stated sufficient conditions for this transformation to be possible: *measPureSpec*, *measJoinSpec*, *measPlusSpec*. We also have seen the different computational patterns that the two implementations exhibit: While *val′* first computes all possible trajectories for the given policy sequence and initial state, then computes their individual sum of rewards and finally applies the measure once, *val* computes its final result by adding the current reward to an intermediate outcome and applying the measure locally at each decision step. This suggests that a transformation from *val′* to *val* will essentially have to push the application of the measure into the recursive computation of the sum of rewards. The proof will be carried out by induction on the structure of policy sequences.

6.1 Deterministic Case

To get a first intuition, let's have a look at what the induction step looks like in the deterministic case (i.e. if we fix monad and measure to be identities):

$$\begin{aligned}
&valMeasTotalReward\ (p :: ps)\ x = \\
&\quad (val'\ (p :: ps)\ x) && =\{ \text{ by definition of } val' \}= \\
&\quad (sumR\ ((MkSigma\ x\ y) ** trj\ ps\ x')) && =\{ \text{ by definition of } sumR \}= \\
&\quad (r\ (head\ (trj\ ps\ x')) \oplus val'\ ps\ x') && =\{ \text{ by } headLemma \}= \\
&\quad (r\ x' \oplus val'\ ps\ x') && =\{ \text{ by induction hypothesis } \}= \\
&\quad (r\ x' \oplus val\ ps\ x') && =\{ \text{ by definition of } val \}= \\
&\quad (val\ (p :: ps)\ x) && \square
\end{aligned}$$

where $y = p\ x$, $x' = next\ t\ x\ y$ and $r = reward\ t\ x\ y$. In the proof sketch, we have first applied the definitions of *val′* and *sumR*. Using the fact that in the deterministic case *trj* returns exactly one state-control sequence and that the *head* of any trajectory starting in x' is just x' (let us call the latter *headLemma*), the left hand side of the sum simplifies to $r\ x'$. Its right hand side amounts to $val'\ ps\ x'$ so that we can apply the induction hypothesis. The rest of the proof only relies on definitional equalities. Thus in the deterministic case *val* and *val′* are unconditionally extensionally equal, i.e. without imposing any of the conditions from section 5.

6.2 Lemmas

To prove the general, monadic case, we proceed similarly. This time, however, the situation is complicated by the presence of the abstract monad M. Instead of being able to use the type structure of some concrete monad, we need to leverage on the properties of M, *meas* and $\oplus$ postulated in section 5. To facilitate the main proof, we first prove three lemmas about the interaction of the measure with the monad structure and the $\oplus$-operator on *Val*. Machine-checked proofs are given in the appendices 2, 3 and 4. The monad laws we use are stated in appendix 1.2. In the remainder of this section, we discuss semi-formal versions of the proofs.

Monad algebras. The first lemma allows us to lift and eliminate an application of the monad's *join* operation:

$$\begin{aligned}&\mathit{measAlgLemma} : \{A, B : \mathit{Type}\} \to (f : B \to \mathit{Val}) \to (g : A \to M\,B) \to\\ &\qquad\qquad (\mathit{meas} \circ \mathit{map}\,(\mathit{meas} \circ \mathit{map}\,f \circ g)) \doteq (\mathit{meas} \circ \mathit{map}\,f \circ \mathit{join} \circ \mathit{map}\,g)\end{aligned}$$

The proof of this lemma hinges on the condition *measJoinSpec*. It allows to trade the application of *join* against an application of *map meas*. The rest is just standard reasoning with monad and functor laws, i.e. we use that the functorial map for M preserves composition and that *join* is a natural transformation:

$$\begin{aligned}&\mathit{measAlgLemma}\,f\,g\,ma =\\ &\quad ((\mathit{meas} \circ \mathit{map}\,(\mathit{meas} \circ \mathit{map}\,f \circ g))\,ma) && =\{\text{ by } \mathit{mapPresComp}\ \}=\\ &\quad ((\mathit{meas} \circ \mathit{map}\,(\mathit{meas} \circ \mathit{map}\,f) \circ \mathit{map}\,g)\,ma) && =\{\text{ by } \mathit{mapPresComp}\ \}=\\ &\quad ((\mathit{meas} \circ \mathit{map}\,\mathit{meas} \circ \mathit{map}\,(\mathit{map}\,f) \circ \mathit{map}\,g)\,ma) && =\{\text{ by } \mathit{measJoinSpec}\ \}=\\ &\quad ((\mathit{meas} \circ \mathit{join} \circ \mathit{map}\,(\mathit{map}\,f) \circ \mathit{map}\,g)\,ma) && =\{\text{ by } \mathit{joinNatTrans}\ \}=\\ &\quad ((\mathit{meas} \circ \mathit{map}\,f \circ \mathit{join} \circ \mathit{map}\,g)\,ma) && \Box\end{aligned}$$

This lemma is generic in the sense that it holds for arbitrary Eilenberg-Moore algebras of a monad. Here we prove it for the framework's measure *meas*, but note that in the appendix we prove a generic version that is then appropriately instantiated.

Head/trajectory interaction. The second lemma amounts to a lifted version of *headLemma* in the deterministic case. Mapping *head* onto an M-structure of trajectories computed with *trj* results in an M-structure filled with the initial state of these trajectories; similarly, mapping $(r \circ \mathit{head} \bigoplus s)$ onto $\mathit{trj}\,\mathit{ps}\,x$ for functions r and s of appropriate type is the same as mapping $(\mathit{const}\,(r\,x) \bigoplus s)$ onto $\mathit{trj}\,\mathit{ps}\,x$ (where *const* is the constant function). We can prove

$$\begin{aligned}&\mathit{headTrjLemma} : \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\,t\,n) \to (r : X\,t \to \mathit{Val}) \to\\ &\qquad\qquad (s : \mathit{StateCtrlSeq}\,t\,(S\,n) \to \mathit{Val}) \to (x : X\,t) \to\\ &\qquad\qquad (\mathit{map}\,(r \circ \mathit{head} \bigoplus s) \circ \mathit{trj}\,\mathit{ps})\,x =\\ &\qquad\qquad (\mathit{map}\,(\mathit{const}\,(r\,x) \bigoplus s) \circ \mathit{trj}\,\mathit{ps})\,x\end{aligned}$$

by doing a case split on *ps*. In case $\mathit{ps} = \mathit{Nil}$, the equality holds because the monad's *pure* is a natural transformation and in case $\mathit{ps} = p :: \mathit{ps}'$ because M's functorial *map* preserves composition.

Measure/sum interaction. The third lemma allows us to both commute the measure into the right summand of an $\bigoplus$-sum and to perform the head/trajectory simplification.

$$
\begin{aligned}
\mathit{measSumLemma} : \{t, n : \mathbb{N}\} &\to (\mathit{ps} : \mathit{PolicySeq}\; t\; n) \to \\
&(r : X\; t \to \mathit{Val}) \to \\
&(s : \mathit{StateCtrlSeq}\; t\; (S\; n) \to \mathit{Val}) \to \\
&(\mathit{meas} \circ \mathit{map}\; (r \circ \mathit{head} \bigoplus s) \circ \mathit{trj}\; \mathit{ps}) \doteq \\
&(r \bigoplus \mathit{meas} \circ \mathit{map}\; s \circ \mathit{trj}\; \mathit{ps})
\end{aligned}
$$

Recall that our third condition from section 5, *measPlusSpec*, plays the role of a distributive law and allows us to "factor out" a partially applied sum $(v\oplus)$ for arbitrary $v : \mathit{Val}$. Given that *measPlusSpec* holds, the lemma is provable by simple equational reasoning:

$$
\begin{aligned}
&\mathit{measSumLemma}\; \mathit{ps}\; r\; s\; x' = \\
&\quad ((\mathit{meas} \circ \mathit{map}\; (r \circ \mathit{head} \bigoplus s) \circ \mathit{trj}\; \mathit{ps})\; x') && =\{ \text{ by } \mathit{headTrjLemma} \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\; (\mathit{const}\; (r\; x') \bigoplus s) \circ \mathit{trj}\; \mathit{ps})\; x') && =\{ \text{ by definition of } \bigoplus, \circ \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\; ((\mathit{const}\; (r\; x') \bigoplus \mathit{id}) \circ s) \circ \mathit{trj}\; \mathit{ps})\; x') && =\{ \text{ by } \mathit{mapPresComp} \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\; (\mathit{const}\; (r\; x') \bigoplus \mathit{id}) \circ \mathit{map}\; s \circ \mathit{trj}\; \mathit{ps})\; x') && =\{ \text{ by definition of } \bigoplus \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\; ((r\; x')\oplus) \circ \mathit{map}\; s \circ \mathit{trj}\; \mathit{ps})\; x') && =\{ \text{ by } \mathit{measPlusSpec} \}= \\
&\quad ((((r\; x')\oplus) \circ \mathit{meas} \circ \mathit{map}\; s \circ \mathit{trj}\; \mathit{ps})\; x') && =\{ \text{ by definition of } \bigoplus \}= \\
&\quad ((r \bigoplus \mathit{meas} \circ \mathit{map}\; s \circ \mathit{trj}\; \mathit{ps})\; x') && \square
\end{aligned}
$$

6.3 Correctness of the BJI-value function

With the above lemmas in place, we now prove that *val* is extensionally equal to val'.

Let $t, n : \mathbb{N}$, $\mathit{ps} : \mathit{PolicySeq}\; t\; n$. We prove *valMeasTotalReward* by induction on *ps*.

Base case. We need to show that for all $x : X\; t$, $\mathit{val}'\; \mathit{Nil}\; x = \mathit{val}\; \mathit{Nil}\; x$. The term $\mathit{val}\; \mathit{Nil}\; x$ immediately reduces to *zero* by definition. Opposed to this, we use the fact that *pure* is a natural transformation in order to simplify $\mathit{val}'\; \mathit{Nil}\; x$ to $\mathit{meas}\; (\mathit{pure}\; \mathit{zero})$, and then crucially need the first measure condition *measPureSpec* to be able to conclude that $\mathit{meas}\; (\mathit{pure}\; \mathit{zero}) = \mathit{zero}$ and thus $\mathit{val}'\; \mathit{Nil} \doteq \mathit{val}\; \mathit{Nil}$.

In equational reasoning style: For all $x : X\; t$,

$$
\begin{aligned}
&\mathit{valMeasTotalReward}\; \mathit{Nil}\; x = \\
&\quad (\mathit{val}'\; \mathit{Nil}\; x) && =\{ \text{ by definition of } \mathit{val}' \}= \\
&\quad (\mathit{meas}\; (\mathit{map}\; \mathit{sumR}\; (\mathit{trj}\; \mathit{Nil}\; x))) && =\{ \text{ by definition of } \mathit{trj} \}= \\
&\quad (\mathit{meas}\; (\mathit{map}\; \mathit{sumR}\; (\mathit{pure}\; (\mathit{Last}\; x)))) && =\{ \text{ by } \mathit{pureNatTrans} \}= \\
&\quad (\mathit{meas}\; (\mathit{pure}\; (\mathit{sumR}\; (\mathit{Last}\; x)))) && =\{ \text{ by definition of } \mathit{sumR} \}= \\
&\quad (\mathit{meas}\; (\mathit{pure}\; \mathit{zero})) && =\{ \text{ by } \mathit{measPureSpec} \}= \\
&\quad (\mathit{zero}) && =\{ \text{ by definition of } \mathit{val} \}= \\
&\quad (\mathit{val}\; \mathit{Nil}\; x)
\end{aligned}
$$

Step case. The induction hypothesis (IH) is: for all $x : X\,t$, $val'\,ps\,x = val\,ps\,x$. We have to show that IH implies that for all $p : Policy\,t$ and $x : X\,t$, the equality $val'\,(p :: ps)\,x = val\,(p :: ps)\,x$ holds.
For brevity (and to economize on brackets), let in the following $y = p\,x$, $mx' = next\,t\,x\,y$, $r = reward\,t\,x\,y$, $trjps = trj\,ps$, and $consxy = ((MkSigma\,x\,y) {**})$.

As in the base case, all that has to be done on the val-side of the equation only depends on definitional equality. However it is more involved to bring the val'-side into a form in which the induction hypothesis can be applied. This is where we leverage on the lemmas proved above.

By definition and because map preserves composition, we know that $val'\,(p :: ps)\,x$ is equal to $(meas \circ map\,((r \circ head) \bigoplus sumR))\,(mx' \ggg\!= trjps)$. We use the relation between the monad's $bind$ and $join$ to eliminate the $bind$-operator from the term, resulting in $meas\,(map\,((r \circ head) \bigoplus sumR)\,(join\,(map\,trjps\,mx')))$. Now we can apply the first lemma from above, $measAlgLemma$, to lift and eliminate the $join$ operation. We thus obtain $meas\,(map\,(meas \circ map\,(r \circ head \bigoplus sumR) \circ trjps)\,mx')$.

To commute the measure under the $\bigoplus$ and get rid of the application of $head$, we use our third lemma, $measSumLemma$, resulting in $meas\,(map\,(r \bigoplus meas \circ map\,sumR \circ trjps)\,mx')$. But $meas \circ map\,sumR \circ trjps$ is just $val'\,ps$ and so we can apply the induction hypothesis. The resulting term is equal to $val\,ps\,x$ by definition.
The more detailed equational reasoning proof: [7]

$$
\begin{array}{ll}
\multicolumn{2}{l}{valMeasTotalReward\,(p :: ps)\,x =} \\
\quad (val'\,(p :: ps)\,x) & =\{\text{ by definition of } val' \}= \\
\quad (meas\,(map\,sumR\,(trj\,(p :: ps)\,x))) & =\{\text{ by definition of } trj \}= \\
\quad (meas\,(map\,sumR\,(map\,consxy\,(mx' \ggg\!= trjps)))) & =\{\text{ by } MapPresComp \}= \\
\quad (meas\,(map\,(sumR \circ consxy)\,(mx' \ggg\!= trjps))) & =\{\text{ by definition of } sumR \}= \\
\quad (meas\,(map\,((r \circ head) \bigoplus sumR)\,(mx' \ggg\!= trjps))) & =\{\text{ by } BindJoinSpec \}= \\
\quad (meas\,(map\,((r \circ head) \bigoplus sumR)\,(join\,(map\,trjps\,mx')))) & =\{\text{ by } measAlgLemma \}= \\
\quad (meas\,(map\,(meas \circ map\,(r \circ head \bigoplus sumR) \circ trjps)\,mx')) & =\{\text{ by } measSumLemma \}= \\
\quad (meas\,(map\,(r \bigoplus meas \circ map\,sumR \circ trjps)\,mx')) & =\{\text{ by definition of } val' \}= \\
\quad (meas\,(map\,(r \bigoplus val'\,ps)\,mx')) & =\{\text{ by induction hypothesis }\}= \\
\quad (meas\,(map\,(r \bigoplus val\,ps)\,mx')) & =\{\text{ by definition of } val \}= \\
\quad (val\,(p :: ps)\,x) &
\end{array}
$$

□

Technical remarks. The above proof of $valMeasTotalReward$ omits some technical details that may be uninteresting for a pen and paper proof, but turn out to be crucial in the setting of an intensional type theory – like Idris – where function extensionality does not hold in general. In particular, we have to postulate that the functorial map preserves extensional equality (see appendix 1.2 and (Botta *et al.*, 2020)) for Idris to accept the proof. In fact, most of the reasoning proceeds by replacing functions that are mapped onto

[7] We are very grateful to the anonymous reviewer who suggested an alternative proof for the induction step. The proof presented here is based on his proof, and his suggestions have lead to significantly weaker conditions on the measure and thus a stronger result.

monadic values by other functions that are only extensionally equal. Using that *map* preserves extensional equality allows to carry out such proofs generically without knowledge of the concrete structure of the functor.

6.4 Correctness of monadic backward induction

As corollary, we can now prove the correctness of monadic backward induction, namely that the policy sequences computed by *bi* are optimal with respect to the measured total reward computed by *val′*:

$biOptMeasTotalReward : (t, n : \mathbb{N}) \to GenOptPolicySeq\ val'\ (bi\ t\ n)$

$biOptMeasTotalReward\ t\ n\ ps'\ x =$
  let $vvEqL = sym\ (valMeasTotalReward\ ps'\ x)$ **in**
  let $vvEqR = sym\ (valMeasTotalReward\ (bi\ t\ n)\ x)$ **in**
  let $biOpt\ = biOptVal\ t\ n\ ps'\ x$ **in**
  $replace\ vvEqR\ (replace\ vvEqL\ biOpt)$

7 Conclusion

We have shown that, for measures of uncertainty that fulfill three general compatibility conditions, the monadic backward induction of the framework for specifying and solving finite-horizon, monadic SDPs proposed in (Botta *et al.*, 2017*a*) is correct.

The main result has been proved via the extensional equality of two value functions: 1) the value function of Bellman's dynamic programming (Bellman, 1957) and optimal control theory (Bertsekas, 1995; Puterman, 2014) that is also at the core of the generic backward induction algorithm of (Botta *et al.*, 2017*a*) and 2) the measured total reward function that specifies the objective of decision making in monadic SDPs: the maximization of a measure of the sum of the rewards along the trajectories rooted at the state associated with the first decision.

Our contribution to verified optimal decision making is twofold: On the one hand, we have implemented a machine-checked generalization of the semi-formal results for deterministic and stochastic SDPs discussed in (Bertsekas, 1995, Prop. 1.3.1) and (Puterman, 2014, Theorem 4.5.1.c). As a consequence, we now have a provably correct method for solving deterministic and stochastic sequential decision problems with their canonical measure functions. On the other hand, we have identified three general conditions that are sufficient for the equivalence between the two value functions to hold. The first two conditions are natural compatibility conditions between the measure of uncertainty *meas* and the monadic operations associated with the uncertainty monad M. The third condition is a relationship between *meas*, the functorial map associated with M and the rule for adding rewards $\oplus$. All three conditions have a straightforward category-theoretical interpretation in terms of Eilenberg-Moore algebras (MacLane, 1978, ch. VI.2). As discussed in section 5.3, the three conditions are independent and have non-trivial implications for the measure and the addition function that cannot be derived from the monotonicity condition on *meas* already imposed in (Ionescu, 2009; Botta *et al.*, 2017*a*).

A consequence of this contribution is that we can now compute verified solutions of stochastic sequential decision problems in which the measure of uncertainty is different from the expected value measure. This is important for applications in which the goal of decision making is, for example, of maximizing the value of worst-case outcomes. To the best of our knowledge, the formulation of the compatibility condition and the proof of the equivalence between the two value functions are novel results.

The latter can be employed in a wider context than the one that has motivated our study: in many practical problems in science and engineering, the computation of optimal policies via backward induction (let apart brute-force or gradient methods) is simply not feasible. In these problems one often still needs to generate, evaluate and compare different policies and our result shows under which conditions such evaluation can safely be done via the "fast" value function *val* of standard control theory.

Finally, our contribution is an application of verified, literal programming to optimal decision making: the sources of this document have been written in literal Idris and are available at (Brede & Botta, 2021), where the reader can also find the bare code and some examples. Although the development has been carried out in Idris, it should be readily reproducible in other implementations of type theory like Agda or Coq.

Acknowledgements

The work presented in this paper was motivated by a remark of Marina Martínez Montero who raised the question of the equivalence between *val* and *val′* (and, thus, of the correctness of the Botta *et al.* framework) during an introduction to verified decision making that the authors gave at UCL (Université catholique de Louvain) in 2019. We are especially thankful to Marina for that question!

We are grateful to Jeremy Gibbons, Christoph Kreitz, Patrik Jansson, Tim Richter and to the JFP editors and reviewers, whose comments and recommendations have lead to significant improvements of the original manuscript.

A very special thanks goes to the anonymous reviewer who has suggested both a more straightforward proof of the *val*-*val′* equality and, crucially, weaker conditions on the measure function for the result to hold. This warrants the applicability of the Botta *et al.* framework for verified decision making to a wider class of problems than our original conditions.

The work presented in this paper heavily relies on free software, among others on Coq, Idris, Agda, GHC, git, vi, Emacs, LaTeX and on the FreeBSD and Debian GNU/Linux operating systems. It is our pleasure to thank all developers of these excellent products. This is TiPES contribution No 37. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 820970 (TiPES –Tipping Points in the Earth System, 2019–2023).

Conflicts of Interest

None.

References

Audebaud, P. and Paulin-Mohring, C. (2009) Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* **74**(8):568–589.

Bellman, R. (1957) *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P. and Shreve, S. E. (1996) *Stochastic Optimal Control: The Discrete Time Case*. Athena Scientific.

Bertsekas, D., Nedić, A. and Ozdaglar, A. (2003) *Convex Analysis and Optimization*. Athena Scientific optimization and computation series. Athena Scientific.

Bertsekas, D., P. (1995) *Dynamic Programming and Optimal Control*. Athena Scientific.

Bird, R. (2014) *Thinking Functionally with Haskell*. Cambridge University Press.

Botta, N., Mandel, A., Hofmann, M., Schupp, S. and Ionescu, C. (2013) Mathematical specification of an agent-based model of exchange. *Proceedings of the AISB Convention 2013, "Do-Form: Enabling Domain Experts to use Formalized Reasoning" Symposium*.

Botta, N., Jansson, P. and Ionescu, C. (2018) The impact of uncertainty on optimal emission policies. *Earth System Dynamics* **9**(2):525–542.

Botta, N. (2016–2021) *IdrisLibs*. https://gitlab.pik-potsdam.de/botta/IdrisLibs.

Botta, N., Jansson, P. and Ionescu, C. (2017a) Contributions to a computational theory of policy advice and avoidability. *J. Funct. Program.* **27**:e23.

Botta, N., Jansson, P., Ionescu, C., Christiansen, D. R. and Brady, E. (2017b) Sequential decision problems, dependent types and generic solutions. *Logical Methods in Computer Science* **13**(1).

Botta, N., Brede, N., Jansson, P. and Richter, T. (2020) *Extensional equality preservation and verified generic programming*. Submitted to J. Funct. Prog., https://arxiv.org/abs/2008.02123.

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* **23**(9):552–593.

Brady, E. (2017) *Type-Driven Development in Idris*. Manning Publications Co.

Brede, N. and Botta, N. (2021) *On the Correctness of Monadic Backward Induction*. Git repository.

De Moor, O. (1995) A generic program for sequential decision processes. *PLILPS '95 Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs* pp. 1–23. Springer.

De Moor, O. (1999) Dynamic programming as a software component. *Proc. 3rd WSEAS Int. Conf. Circuits, Systems, Communications and Computers (CSCC 1999)* 4–8.

Diederich, A. (2001) Sequential decision making. Smelser, N. J. and Baltes, P. B. (eds), *International Encyclopedia of the Social & Behavioral Sciences*, pp. 13917–13922. Pergamon.

Erwig, M. and Kollmansberger, S. (2006) Functional Pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.* **16**(1):21–34.

Finus, M., van Ierland, E. and Dellink, R. (2003) *Stability of Climate Coalitions in a Cartel Formation Game*. FEEM Working Paper No. 61.2003.

Gintis, H. (2007) The dynamics of general equilibrium. *Economic Journal* **117**:1280–1309.

Giry, M. (1981) A categorial approach to probability theory. Banaschewski, B. (ed), *Categorical Aspects of Topology and Analysis*. Lecture Notes in Mathematics 915, pp. 68–85. Springer.

Heitzig, J. (2012) *Bottom-Up Strategic Linking of Carbon Markets: Which Climate Coalitions Would Farsighted Players Form?* SSRN Environmental Economics eJournal.

Helm, C. (2003) International emissions trading with endogenous allowance choices. *Journal of Public Economics* **87**:2737–2747.

Ionescu, C. (2009) *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin.

Jacobs, B. (2011) Probabilities, distribution monads, and convex categories. *Theor. Comput. Sci.* **412**(28):3323–3336.

MacLane, S. (1978) *Categories for the Working Mathematician*. 2nd edn. Graduate Texts in Mathematics. Springer.

Puterman, M. L. (2014) *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

TiPES. (2019–2023) *TiPES H2020 Project Website*. https://www.tipes.dk/.
Wadler, P. (2015) Propositions as types. *Commun. ACM* **58**(12):75–84.

Appendices

1 Preliminaries

1.1 General remarks concerning the Idris formalization

- Idris' type checker often struggles with dependencies in implicit arguments. We sometimes use abbreviations to avoid cluttering the proofs with implicit arguments.
- As a standard, we write $(f \circ g \circ h)\, x$ instead of $f\, (g\, (h\, x))$. When this is a problem for the type checker, we use the second notation.
- Functions that are not defined explicitly are from the Idris standard library. Examples are *cong*, *void* and *concat*.
- Proofs in Idris can be implemented by preorder reasoning. I.e. equational reasoning steps of the form

  $$\begin{aligned} &(t_1) =\{\ step\ \}= \\ &(t_2) \qquad \square \end{aligned}$$

  as displayed in this appendix are actual type-checkable implementations of proofs.

1.2 Monad Laws

In the BJI-framework, M is required to be a *container monad* but none of the standard monad laws (Bird, 2014) is required for the verification result to hold. By contrast, to prove our extended correctness result, we need M to be a full-fledged monad. More specifically, we require of the monad M that

- it is equipped with functor and monad operations:

  $$\begin{aligned} &map \quad : \{A, B : Type\} \to (A \to B) \to M\, A \to M\, B \\ &pure \quad : \{A : Type\} \to A \to M\, A \\ &(\gg\!=) : \{A, B : Type\} \to M\, A \to (A \to M\, B) \to M\, B \\ &join \quad : \{A : Type\} \to M\, (M\, A) \to M\, A \end{aligned}$$

- it preserves extensional equality (Botta *et al.*, 2020), identity and composition of arrows:

  $$\begin{aligned} &mapPresEE \quad : \{A, B : Type\} \to (f, g : A \to B) \to f \doteq g \to map\, f \doteq map\, g \\ &mapPresId \quad : \{A : Type\} \to map\, id \doteq id \\ &mapPresComp : \{A, B, C : Type\} \to (f : A \to B) \to (g : B \to C) \to \\ &\qquad\qquad\qquad map\, (g \circ f) \doteq map\, g \circ map\, f \end{aligned}$$

- Its *pure* and *join* operations are natural transformations (see MacLane, 1978, I.4):

  $$\begin{aligned} &pureNatTrans : \{A, B : Type\} \to (f : A \to B) \to map\, f \circ pure \doteq pure \circ f \\ &joinNatTrans \ : \{A, B : Type\} \to (f : A \to B) \to map\, f \circ join \doteq join \circ map\, (map\, f) \end{aligned}$$

  and fulfill the neutrality and associativity axioms:

$$\begin{aligned} &\mathit{pureNeutralLeft} &&: \{A : \mathit{Type}\} \to \mathit{join} \circ \mathit{pure} \doteq \mathit{id} \\ &\mathit{pureNeutralRight} &&: \{A : \mathit{Type}\} \to \mathit{join} \circ \mathit{map}\ \mathit{pure} \doteq \mathit{id} \\ &\mathit{joinAssoc} &&: \{A : \mathit{Type}\} \to \mathit{join} \circ \mathit{map}\ \mathit{join} \doteq \mathit{join} \circ \mathit{join} \end{aligned}$$

Notice that the above specification of the monad operations is not minimal: $(\gg=)$ is not assumed to be implemented in terms of *join* and *map* (or *map* and *join* via $(\gg=)$ and *pure*). Therefore $(\gg=)$ (pronounced "*bind*"), *join* and *map* have to fulfill:

$$\mathit{bindJoinSpec} : \{A, B : \mathit{Type}\} \to (\mathit{ma} : M\ A) \to (f : A \to M\ B) \to (\mathit{ma} \gg= f) = \mathit{join}\ (\mathit{map}\ f\ \mathit{ma})$$

The equality in the axioms is extensional equality, not the type theory's definitional equality:

$$\begin{aligned} &(\doteq) : \{A, B : \mathit{Type}\} \to (f, g : A \to B) \to \mathit{Type} \\ &(\doteq)\ \ \{A\}\ f\ g = (a : A) \to f\ a = g\ a \end{aligned}$$

As Idris does not have function extensionality, not postulating definitional equalities makes the axioms strictly weaker.

The BJI-framework also requires M to be equipped with type-level membership, with a universal quantifier and with a type-valued predicate

$$\mathit{NotEmpty} : \{A : \mathit{Type}\} \to M\ A \to \mathit{Type}$$

For our purposes, the monad operations are moreover required to fulfill the following preservation laws:

- The M-structure obtained from lifting an element into the monad is not empty:

  $$\mathit{pureNotEmpty} : \{A : \mathit{Type}\} \to (a : A) \to \mathit{NotEmpty}\ (\mathit{pure}\ a)$$

- The monad's *map* and *bind* preserve non-emptiness:

  $$\begin{aligned} \mathit{mapPresNotEmpty} : &\{A, B : \mathit{Type}\} \to (f : A \to B) \to (\mathit{ma} : M\ A) \to \\ &\mathit{NotEmpty}\ \mathit{ma} \to \mathit{NotEmpty}\ (\mathit{map}\ f\ \mathit{ma}) \end{aligned}$$

  $$\begin{aligned} \mathit{bindPresNotEmpty} : &\{A, B : \mathit{Type}\} \to (f : A \to M\ B) \to (\mathit{ma} : M\ A) \to \\ &\mathit{NotEmpty}\ \mathit{ma} \to ((a : A) \to \mathit{NotEmpty}\ (f\ a)) \to \mathit{NotEmpty}\ (\mathit{ma} \gg= f) \end{aligned}$$

As discussed in section 5.3, we could have omitted these non-emptiness preservation laws, but instead would have implicitly restricted the class of monads for which the correctness result holds.

1.3 Preservation of extensional equality

We have stated above that for our correctness proof the functor M has to satisfy the monad laws and moreover its functorial *map* has to preserve extensional equality.

This e.g. is the case for $M = \mathit{Identity}$ (no uncertainty), $M = \mathit{List}$ (non-deterministic uncertainty) and for the finite probability distributions (stochastic uncertainty, $M = \mathit{Prob}$) discussed in (Botta *et al.*, 2017*a*). For $M = \mathit{List}$ the proof of *mapPresEE* amounts to:

$$\begin{aligned} &\mathit{mapPresEE} : \{A, B : \mathit{Type}\} \to (f, g : A \to B) \to f \doteq g \to \mathit{map}\ f \doteq \mathit{map}\ g \\ &\mathit{mapPresEE}\ f\ g\ p\ \ \mathit{Nil} \qquad = \mathit{Refl} \end{aligned}$$

```
mapPresEE f g p (a :: as) =
  (map f (a :: as))  ={ Refl }=
  (f a :: map f as)  ={ cong {f = λx ⇒ x :: map f as} (p a) }=
  (g a :: map f as)  ={ cong (mapPresEE f g p as) }=
  (g a :: map g as)  ={ Refl }=
  (map g (a :: as))                                          □
```

The principle of extensional equality preservation is discussed in more detail in (Botta *et al.*, 2020).

2 Correctness of the value function

```
valMeasTotalReward : {t, n : ℕ} → (ps : PolicySeq t n) → (x : X t) →
                     val′ ps x = val ps x

valMeasTotalReward Nil x =
  (val′ Nil x)                             ={ Refl }=
  (meas (map sumR (trj Nil x)))            ={ Refl }=
  (meas (map sumR (pure (Last x))))        ={ cong (pureNatTrans sumR (Last x)) }=
  (meas (pure (sumR (Last x))))            ={ Refl }=
  (meas (pure zero))                       ={ measPureSpec zero }=
  (zero)                                   ={ Refl }=
  (val Nil x)                                                                   □

valMeasTotalReward {t} {n = S m} (p :: ps) x =
    -- type abbreviations
  let SCS       : Type               = StateCtrlSeq (S t) (S m)           in
  let SCS′      : Type               = StateCtrlSeq t (S (S m))           in
    -- element and function abbreviations
  let y         : Y t x              = p x                                in
  let mx′       : M (X (S t))        = next t x y                         in
  let r         :(X (S t) → Val)     = reward t x y                       in
  let trjps     :(X (S t) → M SCS)   = trj ps                             in
  let consxy    :(SCS → SCS′)        = ((MkSigma x y)∗∗)                  in
  let mx′trjps  : M SCS              = (mx′ ≫= trjps)                     in
  let sR        :(SCS → Val)         = sumR {t = S t} {n = S m}           in
  let hd        :(SCS → X (S t))     = head {t = S t} {n = m}             in
    -- proof steps:
  let useMapPresComp    = mapPresComp consxy sumR mx′trjps                in
  let useBindJoinSpec   = bindJoinSpec {B = SCS} trjps mx′                in
  let useAlgLemma       = algLemma {B = SCS}
                            meas measJoinSpec
                            ((r ∘ hd) ⨁ sumR) trjps mx′                   in
  let useMeasSumLemma   = mapPresEE
                            (meas ∘ map (r ∘ hd ⨁ sR) ∘ trjps)
                            (r ⨁ meas ∘ map sR ∘ trjps)
                            (measSumLemma ps r sR)
                            mx′                                           in
  let useIH             = mapPresEE
```

$$
\begin{array}{l}
\quad (r \bigoplus \mathit{val}' \, \mathit{ps}) \\
\quad (r \bigoplus \mathit{val} \, \mathit{ps}) \\
\quad (\mathit{oplusLiftEERight} \, (\mathit{val}' \, \mathit{ps}) \\
\qquad (\mathit{val} \, \mathit{ps}) \; r \; (\mathit{valMeasTotalReward} \, \mathit{ps})) \\
\qquad \mathit{mx}' \qquad \mathbf{in} \\
\mathbf{let}\; \mathit{ctx} = \lambda a \Rightarrow \mathit{meas} \, (\mathit{map} \, ((r \circ \mathit{hd}) \bigoplus \mathit{sumR}) \; a) \; \mathbf{in}
\end{array}
$$

$$
\begin{array}{ll}
(\mathit{val}' \, (p :: \mathit{ps}) \, x) & =\{ \mathit{Refl} \}= \\
(\mathit{meas} \, (\mathit{map} \, \mathit{sumR} \, (\mathit{trj} \, (p :: \mathit{ps}) \, x))) & =\{ \mathit{Refl} \}= \\
(\mathit{meas} \, (\mathit{map} \, \mathit{sumR} \, (\mathit{map} \, \mathit{consxy} \, \mathit{mx}' \mathit{trjps}))) & =\{ \mathit{cong} \, (\mathit{sym} \, \mathit{useMapPresComp}) \}= \\
(\mathit{meas} \, (\mathit{map} \, (\mathit{sumR} \circ \mathit{consxy}) \, \mathit{mx}' \mathit{trjps})) & =\{ \mathit{Refl} \}= \\
(\mathit{meas} \, (\mathit{map} \, ((r \circ \mathit{hd}) \bigoplus \mathit{sR}) \, \mathit{mx}' \mathit{trjps})) & =\{ \mathit{cong} \, \{ f = \mathit{ctx} \} \, \mathit{useBindJoinSpec} \}= \\
(\mathit{meas} \, (\mathit{map} \, ((r \circ \mathit{hd}) \bigoplus \mathit{sR}) \, (\mathit{join} \, (\mathit{map} \, \mathit{trjps} \, \mathit{mx}')))) & =\{ \mathit{sym} \, \mathit{useAlgLemma} \}= \\
(\mathit{meas} \, (\mathit{map} \, (\mathit{meas} \circ \mathit{map} \, (r \circ \mathit{hd} \bigoplus \mathit{sR}) \circ \mathit{trjps}) \, \mathit{mx}')) & =\{ \mathit{cong} \, \mathit{useMeasSumLemma} \}= \\
(\mathit{meas} \, (\mathit{map} \, (r \bigoplus \mathit{meas} \circ \mathit{map} \, \mathit{sR} \circ \mathit{trjps}) \, \mathit{mx}')) & =\{ \mathit{Refl} \}= \\
(\mathit{meas} \, (\mathit{map} \, (r \bigoplus \mathit{val}' \, \mathit{ps}) \, \mathit{mx}')) & =\{ \mathit{cong} \, \mathit{useIH} \}= \\
(\mathit{meas} \, (\mathit{map} \, (r \bigoplus \mathit{val} \, \mathit{ps}) \, \mathit{mx}')) & =\{ \mathit{Refl} \}= \\
(\mathit{val} \, (p :: \mathit{ps}) \, x) & \square
\end{array}
$$

3 Correctness of monadic backward induction

Together with the result of Botta *et al.* (*biOptVal*, see appendix 6 below) we can prove the correctness of monadic backward induction as corollary, using a generalized optimality of policy sequences predicate:

$$
\begin{array}{l}
\mathit{GenOptPolicySeq} : \{ t, n : \mathbb{N} \} \to (f : \mathit{PolicySeq} \; t \; n \to X \; t \to \mathit{Val}) \to \mathit{PolicySeq} \; t \; n \to \mathit{Type} \\
\mathit{GenOptPolicySeq} \; \{ t \} \; \{ n \} \; f \; \mathit{ps} = (\mathit{ps}' : \mathit{PolicySeq} \; t \; n) \to (x : X \; t) \to f \; \mathit{ps}' \; x \sqsubseteq f \; \mathit{ps} \; x
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{biOptMeasTotalReward} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{GenOptPolicySeq} \; \mathit{val}' \; (\mathit{bi} \; t \; n)} \\
\multicolumn{3}{l}{\mathit{biOptMeasTotalReward} \; t \; n \; \mathit{ps}' \; x =} \\
\quad \mathbf{let}\; \mathit{vvEqL} = \mathit{sym} \, (\mathit{valMeasTotalReward} \, \mathit{ps}' \, x) & \mathbf{in} & \text{-- } \mathit{val}' \, \mathit{ps}' \, x = \mathit{val} \, \mathit{ps}' \, x \\
\quad \mathbf{let}\; \mathit{vvEqR} = \mathit{sym} \, (\mathit{valMeasTotalReward} \, (\mathit{bi} \; t \; n) \, x) & \mathbf{in} & \text{-- } \mathit{val}' \, (\mathit{bi} \; t \; n) \, x = \mathit{val} \, (\mathit{bi} \; t \; n) \, x \\
\quad \mathbf{let}\; \mathit{biOpt} \; = \mathit{biOptVal} \; t \; n \; \mathit{ps}' \; x & \mathbf{in} & \text{-- } \mathit{val} \, \mathit{ps}' \, x \sqsubseteq \mathit{val} \, (\mathit{bi} \; t \; n) \, x \\
\quad \mathbf{let}\; \mathit{lP} \;\;\; = \lambda v \Rightarrow v \sqsubseteq \mathit{val} \, (\mathit{bi} \; t \; n) \, x & \mathbf{in} & \\
\quad \mathbf{let}\; \mathit{rP} \;\;\; = \lambda v \Rightarrow \mathit{val}' \, \mathit{ps}' \, x \sqsubseteq v & \mathbf{in} & \\
\multicolumn{3}{l}{\quad \mathit{replace} \; \{ P = \mathit{rP} \} \; \mathit{vvEqR} \; (\mathit{replace} \; \{ P = \mathit{lP} \} \; \mathit{vvEqL} \; \mathit{biOpt})}
\end{array}
$$

4 Lemmas

The proof of *valMeasTotalReward* relies on a few auxiliary results which we prove here.

Lemma about the interaction of *head* and *trj* interleaved with *map*:

$$
\begin{array}{l}
\mathit{headTrjLemma} : \{ t, n : \mathbb{N} \} \to (\mathit{ps} : \mathit{PolicySeq} \; t \; n) \to \\
\qquad (r : X \; t \to \mathit{Val}) \to (s : \mathit{StateCtrlSeq} \; t \; (S \; n) \to \mathit{Val}) \to (x' : X \; t) \to \\
\qquad (\mathit{map} \, (r \circ \mathit{head} \bigoplus s) \circ \mathit{trj} \, \mathit{ps}) \; x' = (\mathit{map} \, (\mathit{const} \, (r \; x') \bigoplus s) \circ \mathit{trj} \, \mathit{ps}) \; x'
\end{array}
$$

```
headTrjLemma {t} {n = Z} Nil r s x′ =
  let hd       = head {t}                                           in
  let lastx′   = Last {t} x′                                        in
  let nil      = Nil {t}                                            in
  let usePureNatTrans    = pureNatTrans (r ∘ hd ⨁ s) lastx′          in
  let usePureNatTransSym = sym (pureNatTrans ((⊕) (r x′) ∘ s) lastx′) in
    ((map (r ∘ hd ⨁ s)) (trj nil x′))          ={ Refl }=
    ((map (r ∘ hd ⨁ s)) (pure lastx′))         ={ usePureNatTrans }=
    ((pure ∘ (r ∘ hd ⨁ s)) lastx′)             ={ Refl }=
    ((pure ∘ (const (r x′) ⨁ s)) lastx′)       ={ usePureNatTransSym }=
    (map (const (r x′) ⨁ s) (pure lastx′))     ={ Refl }=
    (map (const (r x′) ⨁ s) (trj nil x′))                                  □
headTrjLemma {t = t} {n = S n} (p :: ps) r s x′ =
  let y′       = p x′                                                      in
  let mx″      = next t x′ y′                                              in
  let SCS = (StateCtrlSeq (S t) (S n))                                     in
  let consx′y′ = (∗∗) {t} {n} (MkSigma x′ y′)                              in
  let mx″trjps = (≫=) {B = SCS} mx″ (trj ps)                               in
  let traj     = trj {t}                                                   in
  let useMapPresComp    = mapPresComp consx′y′ (const (r x′) ⨁ s) mx″trjps in
  let useMapPresCompSym = sym (mapPresComp consx′y′ (r ∘ head ⨁ s) mx″trjps) in
    ((map (r ∘ head ⨁ s)) (traj (p :: ps) x′))            ={ Refl }=
    ((map (r ∘ head ⨁ s) ∘ map consx′y′) mx″trjps)        ={ useMapPresCompSym }=
    ((map ((r ∘ head ⨁ s) ∘ consx′y′)) mx″trjps)          ={ Refl }=
    (map ((const (r x′) ⨁ s) ∘ consx′y′) mx″trjps)        ={ useMapPresComp }=
    (map (const (r x′) ⨁ s) (map consx′y′ mx″trjps))      ={ Refl }=
    (map (const (r x′) ⨁ s) (traj (p :: ps) x′))                           □
```

Lemma about the commutation of *meas* ∘ *map* and ⨁:

```
measSumLemma : {t, n : ℕ} → (ps : PolicySeq t n) →
               (r : X t → Val) →
               (s : StateCtrlSeq t (S n) → Val) →
               (meas ∘ map (r ∘ head ⨁ s) ∘ trj ps) ≐
               (r ⨁ meas ∘ map s ∘ trj ps)

measSumLemma {t} {n} ps r s x′ =
    -- non-emptiness proofs
  let trjNE            = trjNotEmptyLemma ps x′                        in
  let sNE              = mapPresNotEmpty s (trj ps x′) trjNE           in
    -- proof steps
  let useMeasPlusSpec  = measPlusSpec (r x′) (map s (trj ps x′)) sNE   in
  let useMapPresComp   = cong {f = \ prf ⇒ meas prf}
                           (mapPresComp s ((r x′)⊕) (trj ps x′))       in
  let useHdTrjLemma    = cong (headTrjLemma ps r s x′)                 in
  ((meas ∘ map (r ∘ head ⨁ s) ∘ trj ps) x′)            ={ useHdTrjLemma }=
  ((meas ∘ map (const (r x′) ⨁ s) ∘ trj ps) x′)        ={ Refl }=
  ((meas ∘ map ((const (r x′) ⨁ id) ∘ s) ∘ trj ps) x′) ={ useMapPresComp }=
```

$$
\begin{aligned}
&((meas \circ map\ (const\ (r\ x') \bigoplus id) \circ map\ s \circ trj\ ps)\ x') &&=\{\ Refl\ \}= \\
&((meas \circ map\ ((r\ x') \oplus) \circ map\ s \circ trj\ ps)\ x') &&=\{\ useMeasPlusSpec\ \}= \\
&((((r\ x') \oplus) \circ meas \circ map\ s \circ trj\ ps)\ x') &&=\{\ Refl\ \}= \\
&(((const\ (r\ x') \bigoplus id) \circ meas \circ map\ s \circ trj\ ps)\ x') &&=\{\ Refl\ \}= \\
&((r \bigoplus meas \circ map\ s \circ trj\ ps)\ x') && \Box
\end{aligned}
$$

The *trj* function never produces an empty M-structure of trajectories:

$$trjNotEmptyLemma : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to NotEmpty\ (trj\ ps\ x)$$

$$
\begin{aligned}
&trjNotEmptyLemma\ (Nil) \qquad x = pureNotEmpty\ (Last\ x) \\
&trjNotEmptyLemma\ \{t = t\}\ \{n = S\ n\}\ (p :: ps)\ x = \\
&\quad \mathbf{let}\ y = p\ x\ \mathbf{in} \\
&\quad \mathbf{let}\ trjps = trj\ ps\ \mathbf{in} \\
&\quad \mathbf{let}\ nxpx = next\ t\ x\ y\ \mathbf{in} \\
&\quad \mathbf{let}\ consxy = ((MkSigma\ x\ y) {**})\ \mathbf{in} \\
&\quad \mathbf{let}\ nne = nextNotEmpty\ x\ y\ \mathbf{in} \\
&\quad \mathbf{let}\ netrjps = trjNotEmptyLemma\ ps\ \mathbf{in} \\
&\quad \mathbf{let}\ bne = bindPresNotEmpty\ trjps\ nxpx\ nne\ netrjps\ \mathbf{in} \\
&\quad mapPresNotEmpty\ consxy\ (nxpx \ggg\!= trjps)\ bne
\end{aligned}
$$

A technical lemma to lift equalities into the right component of $\bigoplus$:

$$oplusLiftEERight : \{A : Type\} \to (f, g, h : A \to Val) \to (g \doteq h) \to (f \bigoplus g) \doteq (f \bigoplus h)$$

$$oplusLiftEERight\ \{A\}\ f\ g\ h\ ee\ a = cong\ (ee\ a)$$

4.1 Properties of monad algebras

Condition for a function f to be an M-algebra homomorphism:

$$
\begin{aligned}
&algMorSpec : \{A, B : Type\} \to (\alpha : M\ A \to A) \to (\beta : M\ B \to B) \to (f : A \to B) \to Type \\
&algMorSpec\ \{A\}\ \{B\}\ \alpha\ \beta\ f = (\beta \circ map\ f) \doteq (f \circ \alpha)
\end{aligned}
$$

Structure maps of M-algebras are left inverses of *pure*:

$$
\begin{aligned}
&algPureSpec : \{A : Type\} \to (\alpha : M\ A \to A) \to Type \\
&algPureSpec\ \{A\}\ \alpha = \alpha \circ pure \doteq id
\end{aligned}
$$

Structure maps of M-algebras are themselves M-algebra homomorphisms:

$$
\begin{aligned}
&algJoinSpec : \{A : Type\} \to (\alpha : M\ A \to A) \to Type \\
&algJoinSpec\ \{A\}\ \alpha = algMorSpec\ join\ \alpha\ \alpha \quad \text{-- } (\alpha \circ map\ \alpha) \doteq (\alpha \circ join)
\end{aligned}
$$

A lemma about computation with M-algebras:

$$
\begin{aligned}
algLemma : &\{A, B, C : Type\} \to (\alpha : M\ C \to C) \to (ee : algJoinSpec\ \alpha) \to \\
&(f : B \to C) \to (g : A \to M\ B) \to \\
&(\alpha \circ map\ (\alpha \circ map\ f \circ g)) \doteq (\alpha \circ map\ f \circ join \circ map\ g)
\end{aligned}
$$

$$
\begin{aligned}
&algLemma\ \{A\}\ \{B\}\ \{C\}\ \alpha\ ee\ f\ g\ ma = \\
&\quad ((\alpha \circ map\ (\alpha \circ map\ f \circ g))\ ma) &&=\{\ cong\ (mapPresComp\ g\ (\alpha \circ map\ f)\ ma)\ \}= \\
&\quad ((\alpha \circ map\ (\alpha \circ map\ f) \circ map\ g)\ ma) &&=\{\ cong\ (mapPresComp\ (map\ f)\ \alpha\ (map\ g\ ma))\ \}=
\end{aligned}
$$

$$
\begin{array}{ll}
((\alpha \circ map\ \alpha \circ map\ (map f) \circ map\ g)\ ma) & =\{\ ee\ (map\ (map f)\ (map\ g\ ma))\ \}= \\
((\alpha \circ join \circ map\ (map f) \circ map\ g)\ ma) & =\{\ cong\ (sym\ (joinNatTrans f\ (map\ g\ ma)))\ \}= \\
((\alpha \circ map f \circ join \circ map\ g)\ ma) & \square
\end{array}
$$

4.2 Measure specifications

. The measure needs to be the structure map of an M-algebra on *Val*. This means:

- It is a left inverse to *pure*:

  $measPureSpec : algPureSpec\ meas \quad \text{-- } meas \circ pure \doteq id$

- It is an M-algebra homomorphism from *join* to itself:

  $measJoinSpec : algJoinSpec\ meas \quad \text{-- } meas \circ join \doteq meas \circ map\ meas$

Moreover, for all $v : Val$, the function $(v\oplus)$ needs to be an M-algebra homomorphism:

$$
\begin{array}{l}
measPlusSpec : (v : Val) \to (mv : M\ Val) \to (NotEmpty\ mv) \to \\
\qquad (meas \circ map\ (v\oplus))\ mv = ((v\oplus) \circ meas)\ mv
\end{array}
$$

We can omit the non-emptiness condition but this means we implicitly restrict the class of monads that can be used for M. The condition could then be expressed as

$$measPlusSpec' : (v : Val) \to algMorSpec\ meas\ meas\ (\oplus v)$$

5 Bellman's principle of optimality

Basic requirements for monadic backward induction:

$$(\sqsubseteq) : Val \to Val \to Type$$

$$
\begin{array}{ll}
lteRefl & : \{a : Val\} \to a \sqsubseteq a \\
lteTrans & : \{a, b, c : Val\} \to a \sqsubseteq b \to b \sqsubseteq c \to a \sqsubseteq c \\
plusMonSpec & : \{a, b, c, d : Val\} \to a \sqsubseteq b \to c \sqsubseteq d \to (a \oplus c) \sqsubseteq (b \oplus d)
\end{array}
$$

$$
\begin{array}{l}
measMonSpec : \{A : Type\} \to (f, g : A \to Val) \to ((a : A) \to (f\ a) \sqsubseteq (g\ a)) \to \\
\qquad (ma : M\ A) \to meas\ (map f\ ma) \sqsubseteq meas\ (map\ g\ ma)
\end{array}
$$

Optimality of policy sequences:

$$
\begin{array}{l}
OptPolicySeq : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to Type \\
OptPolicySeq\ \{t\}\ \{n\}\ ps = (ps' : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps'\ x \sqsubseteq val\ ps\ x
\end{array}
$$

Optimality of extensions of policy sequences:

$$
\begin{array}{l}
OptExt : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t \to Type \\
OptExt\ \{t\}\ ps\ p = (p' : Policy\ t) \to (x : X\ t) \to val\ (p' :: ps)\ x \sqsubseteq val\ (p :: ps)\ x
\end{array}
$$

Bellman's principle of optimality:

$$
\begin{array}{llll}
Bellman : \{t, n : \mathbb{N}\} \to & (ps : PolicySeq\ (S\ t)\ n) & \to OptPolicySeq\ ps & \to \\
 & (p\ \ : Policy\ t) & \to OptExt\ ps\ p & \to OptPolicySeq\ (p :: ps)
\end{array}
$$

```
Bellman {t} ps ops p oep (p′ :: ps′) x =
  let y′  = p′ x                                    in
  let mx′ = next t x y′                             in
  let f′  = reward t x y′ ⊕ val ps′                 in
  let f   = reward t x y′ ⊕ val ps                  in
  let s₀  = λx′ ⇒ plusMonSpec lteRefl (ops ps′ x′)  in
  let s1  = measMonSpec f′ f s₀ mx′                 in   -- val (p′ :: ps′) x ⊑ val (p′ :: ps) x
  let s2  = oep p′ x                                in   -- val (p′ :: ps) x ⊑ val (p :: ps) x
  lteTrans s1 s2
```

6 Verification with respect to *val*

The empty policy sequence is optimal:

```
nilOptPolicySeq : OptPolicySeq Nil
nilOptPolicySeq Nil x = lteRefl
```

Now, provided that we can implement

```
optExt     : {t, n : ℕ} → PolicySeq (S t) n → Policy t
optExtSpec : {t, n : ℕ} → (ps : PolicySeq (S t) n) → OptExt ps (optExt ps)
```

then

```
bi : (t : ℕ) → (n : ℕ) → PolicySeq t n
bi t Z     = Nil
bi t (S n) = let ps = bi (S t) n in optExt ps :: ps
```

is correct with respect to *val*:

```
biOptVal : (t : ℕ) → (n : ℕ) → OptPolicySeq (bi t n)

biOptVal t Z     = nilOptPolicySeq
biOptVal t (S n) =
  let ps  = bi (S t) n       in
  let ops = biOptVal (S t) n in
  let p   = optExt ps        in
  let oep = optExtSpec ps    in
  Bellman ps ops p oep
```

7 Optimal extension

The generic implementation of backward induction *bi* naturally raises the question under which conditions one can implement *optExt* such that *optExtSpec* holds.
To this end, consider the function

```
cval : {t, n : ℕ} → PolicySeq (S t) n → (x : X t) → Y t x → Val
cval {t} ps x y = let mx′ = next t x y in
                  meas (map (reward t x y ⊕ val ps) mx′)
```

By definition of *val* and *cval*, one has

$$
\begin{array}{l}
\mathit{val}\;(p :: \mathit{ps})\;x \\
\quad = \\
\mathit{meas}\;(\mathit{map}\;(\mathit{reward}\;t\;x\;(p\;x) \oplus \mathit{val}\;\mathit{ps})\;(\mathit{next}\;t\;x\;(p\;x))) \\
\quad = \\
\mathit{cval}\;\mathit{ps}\;x\;(p\;x)
\end{array}
$$

This suggests that, if we can maximize *cval* that is, implement

$$
\begin{array}{ll}
\mathit{cvalmax} & : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\;(S\;t)\;n \to (x : X\;t) \to \mathit{Val} \\
\mathit{cvalargmax} & : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\;(S\;t)\;n \to (x : X\;t) \to Y\;t\;x
\end{array}
$$

that fulfill

$$
\begin{array}{ll}
\mathit{cvalmaxSpec} : & \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\;(S\;t)\;n) \to (x : X\;t) \to \\
& (y : Y\;t\;x) \to \mathit{cval}\;\mathit{ps}\;x\;y \sqsubseteq \mathit{cvalmax}\;\mathit{ps}\;x \\
\mathit{cvalargmaxSpec} : & \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\;(S\;t)\;n) \to (x : X\;t) \to \\
& \quad \mathit{cvalmax}\;\mathit{ps}\;x = \mathit{cval}\;\mathit{ps}\;x\;(\mathit{cvalargmax}\;\mathit{ps}\;x)
\end{array}
$$

then we can implement optimal extensions of arbitrary policy sequences. As it turns out, this intuition is correct. With

$$\mathit{optExt} = \mathit{cvalargmax}$$

one has

$$
\begin{array}{l}
\mathit{optExtSpec}\;\{t\}\;\{n\}\;\mathit{ps}\;p'\;x = \\
\quad \mathbf{let}\;p \;\; = \mathit{optExt}\;\mathit{ps} \qquad\qquad \mathbf{in} \\
\quad \mathbf{let}\;y \;\; = p\;x \qquad\qquad\qquad \mathbf{in} \\
\quad \mathbf{let}\;y' = p'\;x \qquad\qquad\qquad \mathbf{in} \\
\quad \mathbf{let}\;s1 = \mathit{cvalmaxSpec}\;\mathit{ps}\;x\;y'\;\mathbf{in} \\
\quad \mathbf{let}\;s2 = \mathit{replace}\;\{P = \lambda z \Rightarrow (\mathit{cval}\;\mathit{ps}\;x\;y' \sqsubseteq z)\}\;(\mathit{cvalargmaxSpec}\;\mathit{ps}\;x)\;s1\;\mathbf{in} \\
\quad s2
\end{array}
$$

The observation that functions *cvalmax* and *cvalargmax* that fulfill *cvalmaxSpec* and *cvalargmaxSpec* are sufficient to implement an optimal extension *optExt* that fulfills *optExtSpec* naturally raises the question of what are necessary and sufficient conditions for *cvalmax* and *cvalargmax*. Answering this question necessarily requires discussing properties of *cval* and goes well beyond the scope of formulating a theory of SDPs. Here, we limit ourselves to remark that if $Y\;t\;x$ is finite and non-empty one can implement the functions *cvalmax* and *cvalargmax* by linear search. A generic implementation of *cvalmax* and *cvalargmax* can be found under (Brede & Botta, 2021).

For the original BJI-theory, tabulated backward induction and several example applications can be found in the *SequentialDecisionProblems* folder of (Botta, 2016–2021).